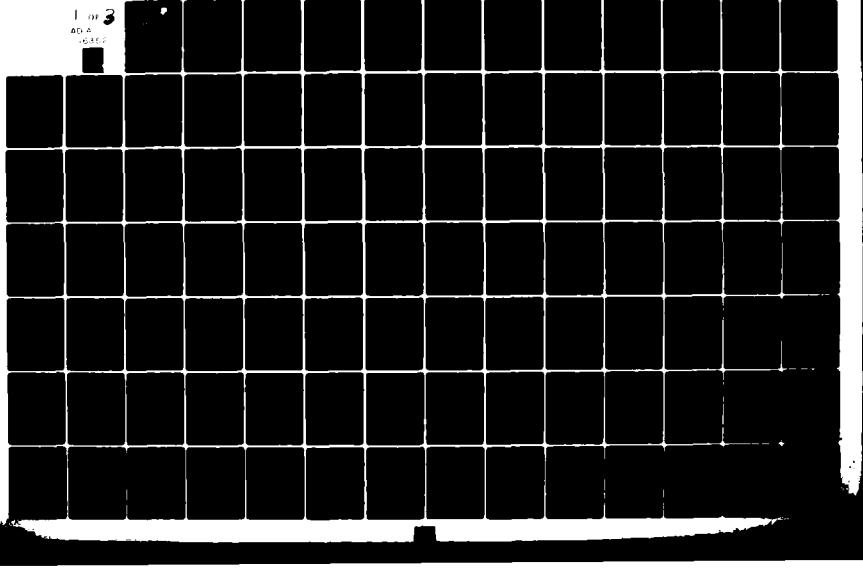
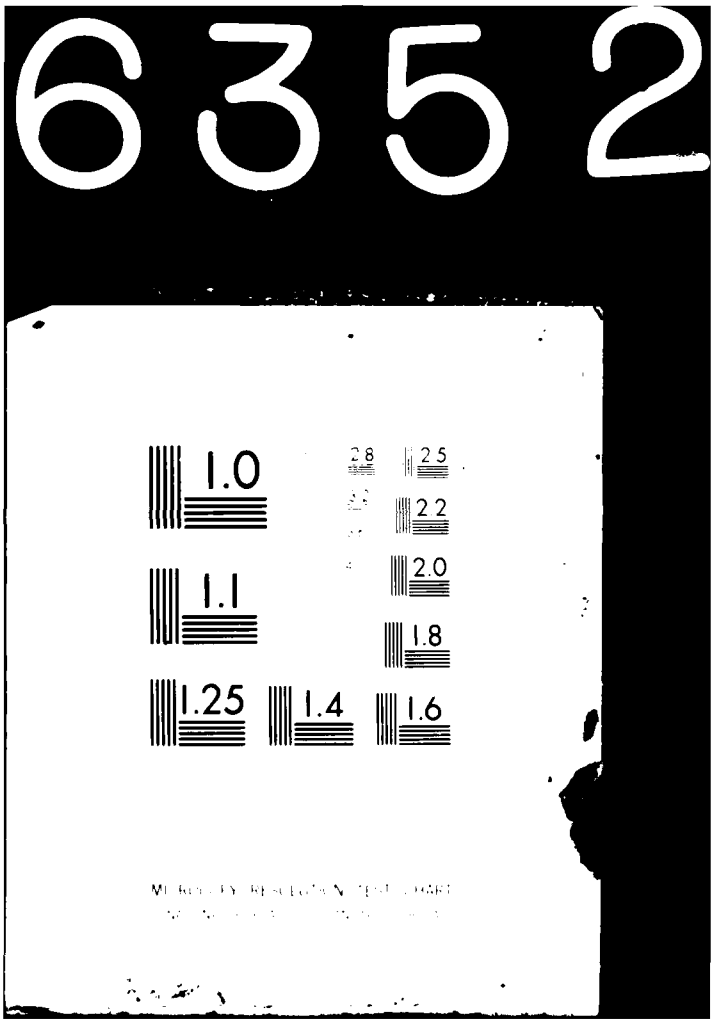
AD-A116 352 SOFTWARE ENGINEERING ASSOCIATES INC TORRANCE CA F/G 9/2
JOVIAL/ADA MICROPROCESSOR STUDY.(U)
APR 82 T E DEVINE, T L DUNBAR, M B LITTLEJOHN F30602-80-C-0153
UNCLASSIFIED RADC-TR-82-61 NL




6352
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5

RADC-TR-82-61
Final Technical Report
April 1982

JOVIAL/ADA MICROPROCESSOR STUDY

Software Engineering Associates, Inc.

Terence E. Devine, Terry L. Dunbar, Michael B. Littlejohn and Kerry White

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) an is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-82-61 has been reviewed and is approved for publication.

APPROVED: Richard M. Motto

RICHARD M. MOTTO
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-82-61	2. GOVT ACCESSION NO. AD-A113 352 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): JOVIAL/ADA MICROPROCESSOR STUDY	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, April 80 - November 1981
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Terence E. Devine, Michael B. Littlejohn, Terry L. Dunbar, Kerry White	8. CONTRACT OR GRANT NUMBER(s): F30602-80-C-0153
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Software Engineering Associates Inc., 23864 Hawthorne Blvd., Suite 200, Torrance CA 90505	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 5581910
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (COES), Griffiss AFB NY 13441	12. REPORT DATE: April 1982
	13. NUMBER OF PAGES: 202
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Richard M. Motto (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microcomputers
Ada
Microprocessors
Software Systems
Software Tools
Development Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The initial intent of this effort was to study the Microprocessor industry for a period of eighteen months and to configure a Microprocessor System capable of hosting an Ada compiler or at least a JOVIAL/J73 compiler. Shortly into the study, the JOVIAL portion of the effort was eliminated.

Basically, the study investigated the following areas:

a. Current and future microprocessors technology,

b. Existing compilers on small hosts,

(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

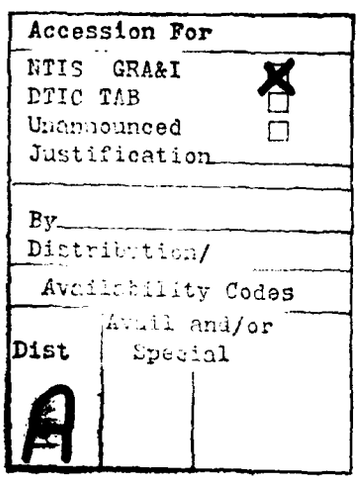
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

c. Advantages and disadvantages of hosting a development system on a microprocessor,
d. Ada language issues,
e. The design of an Ada Integrated Encironment (AIE) for a microprocessor

The study concluded that a workable Ada capability can be hosted on a Motorola MC68000, Zilog Z8000, Intel 8086, Intel 432 or the soon-to-be-marketed National Semiconductor 16032. A practical micro-based system would support one of the above mentioned microprocessor, a terminal (CRT or hand copy), floppy disks, a Winchester disk, applicable disk controllers and 256 Bytes of RAM. Obviously, each application directs the particular hardware required to satisfy that application. At the time of this report, several manufactors were advertising partial implementations of Ada hosted on a variety of hardware configurations.

ACKNOWLEDGEMENTS

The authors would like to thank Dr. Ken Bowles and Richard Kaufmann of TeleSoft for their help in running benchmarks of the TeleSoft Ada compiler.



INDEX

Appendices

JOVIAL/Ada Microprocessor Study Final Technical Report

1. INTRODUCTION

The purpose of this study was to determine if a microcomputer could host a software development system which provided, among other tools, JOVIAL J73 or Ada compilers. At the time the study was envisioned, there was some question as to whether satisfactory performance could be obtained with the then current hardware. Consequently, the study was to include a projection of the hardware which was expected to be available by 1985. In addition, a design for a microcomputer-based development system was to be presented, along with estimates of its performance on present-day and 1985 hardware.

The field of microcomputers is changing so rapidly that almost anything which is written about the current state of the field is outdated by the time it is published. In the time since the initiation of this study, an Ada development system hosted on a micro has appeared on the market. Thus, the question is not now one of "Can a development system be hosted on a microcomputer?", but rather "What should such a system look like and how many users can it support?"

The material relating to available hardware contained in this report was obtained through a literature search and by contacting various hardware suppliers. Benchmark programs were executed to obtain information about the various machines and implementation techniques. The software system design which is presented here represents the fruits of original research, together with ideas culled from the literature.

1.1 Report Organization

The main body of this report is divided into 12 chapters, which cover such topics as the available hardware, the advantages and disadvantages of microprocessor-based systems, performance results, a high level look at a system design, and estimates of its performance. The detailed specifications are presented in an abbreviated B5 specification format (the testing and quality assurance sections are omitted) in separate volumes.

The body of the report is organized into chapters as follows:

Chapter 1 (this chapter) provides an introduction to the report.

Chapter 2 provides background information.

Chapter 3 discusses current and future microprocessor technology.

Chapter 4 discusses the advantages of a microcomputer-hosted development system as compared with a system on a large host.

Chapter 5 discusses the disadvantages of hosting a development system on a microcomputer.

Chapter 6 discusses the prospects for hosting a system on a micro.

Chapter 7 discusses existing compilers, both on large and small hosts, and their performance.

Chapter 8 discusses Ada language issues.

Chapter 9 discusses the minimum capabilities that are necessary to have a useful development system.

Chapter 10 gives an overview of our proposed development system.

Chapter 11 discusses the operating system.

Chapter 12 discusses the database, including the Ada program library.

Chapter 13 discusses the command language interpreter.

Chapter 14 discusses the Ada compiler.

Chapter 15 discusses the linker.

Chapter 16 discusses the source-level debugger.

Chapter 17 discusses the text editor.

Chapter 18 discusses the other tools in the system. These include tools used to develop the system, as well as other useful utilities.

Chapter 19 provides performance estimates for the system, assuming it is hosted on various hardware configurations.

Chapter 20 is the summary of results.

Chapter 21 is the bibliography.

1.2 Acronyms, Terms and Trademarks

APSE	Ada Programming Support Environment
CLI	COMMAND LANGUAGE INTERPRETER
CP/M	A microcomputer operating system (CP/M is a trademark of Digital Research)
DEC-10	A mainframe (DEC is a trademark of Digital Equipment Corp)
DIS	Digital Integrating Subsystem - a 16-bit microprocessor which is basically a Z8002 with memory mapped I/O
DMA	Direct Memory Access - a method of performing I/O in which memory is accessed by the I/O controller without requiring the services of the CPU.
ECL	Emitter Coupled Logic
Horizon	An 8-bit microcomputer (Horizon is a trademark of North Star Computers)
i	A trademark of Intel
ICE	In-Circuit Emulation (ICE is a trademark of Intel)
ISAM	Indexed Sequential Access Method
JCL	Job Control Language
K	2**10 = 1,024
KAPSE	Kernel APSE
KB	Kilobyte (1024 bytes)
LSI	Large Scale Integration
LSI-11	LSI implementation of the PDP-11 (LSI-11 is a trademark of Digital Equipment Corp.)
M	2**20 = 1,048,576
MB	Megabyte (1,048,576 bytes)
MC68000	A 16-bit microprocessor, also known as the 68000 (MC68000 is a trademark of Motorola)

MDS	Microprocessor Development System
Minifloppy	A trademark of Shugart Associates
MOS	Metallic Oxide Semiconductor
MULTIBUS	a standard bus (IEEE 796) (Multibus is a trademark of Intel)
Naked Mini	A Mini/microcomputer family (Naked Mini is a trademark of Computer Automation)
Native Code	Machine code, as opposed to interpretive code.
Nova	A Mini/microcomputer family (Nova is a trademark of Data General Corp.)
Pascal/Z	A Pascal compiler for the Z80 (Pascal/Z is a trademark of Ithaca Intersystems)
P-code	Pseudo-code - In this report the term refers to the interpretable output of the USCD Pascal compiler.
PDP-6	A 36-bit computer (PDP is a trademark of Digital Equipment Corp.)
PDP-8	A 12-bit minicomputer (PDP is a trademark of Digital Equipment Corp.)
PDP-11	A 16-bit minicomputer family
PL/I-80	A PL/I subset compiler (PL/I-80 is a trademark of Digital Research)
Q-Bus	A microcomputer bus (Q-Bus is a trademark of Digital Equipment Corp.)
S-100	A microcomputer bus (IEEE 696)
TI990	A mini/microcomputer family produced by Texas Instruments
UCSD Pascal	A microcomputer Pascal compiler (UCSD is a trademark of the Regents of the University of California)
UNIX	An operating system (UNIX is a trademark of Bell Laboratories)
VAX	A supermini family (VAX is a trademark of Digital Equipment

	Corp.)
VLSI	Very Large Scale Integration
Z80	An 8-bit microprocessor (Z80 is a trademark of Zilog)
Z8000	A 16-bit microprocessor family (Z8000 is a trademark of Zilog)
Z8001	A 16-bit microprocessor with a segmented address space (Z8001 is a trademark of Zilog)
Z8002	A 16-bit microprocessor with a non-segmented address space (Z8002 is a trademark of Zilog)
6502	An 8-bit microprocessor
8080	An 8-bit microprocessor (8080 is a trademark of Intel)
8086	A 16-bit microprocessor (8086 is a trademark of Intel)

2. BACKGROUND

There have been three major cycles in program development since the advent of the electronic computer. The original machines, although physically large, had a fraction of the capabilities of today's mainframes. In fact, the capabilities of early computers are on the order of those found today in programmable calculators [Computer, Jan. 1980, p.5]. Program development was originally done in machine language and later in assembly language. Since the memories of the early machines were small, and since their processors were slow, it was crucial that the program be coded in such a way as to minimize their time and space requirements. Hardware was the principal cost in a computer system.

As machine sizes and speeds increased, it became possible to write programs in higher-level languages, letting the compiler take care of much of the bookkeeping which had previously been the responsibility of the assembly language programmer. Still, there was great concern about the efficiency of the compiled code. One of the concerns of the implementors of the first FORTRAN compiler was that the language would not be accepted if compiled code executed too slowly [Backus78].

In the 1960's a second cycle began as minicomputers arrived on the scene. Again, memories were small and the machines were relatively slow. Assembly language was the primary vehicle for programming these machines. Word sizes of minicomputers tended to be 12 or 16 bits. This provided one of the principal means of distinguishing between mainframes and microcomputers. In addition memories tended to be smaller and the instruction sets more primitive on minicomputers.

Minicomputers also grew, both in terms of memory and word size. High-level languages came into broader use as machine capabilities increased. Currently there is a good deal of overlap between the word sizes of minicomputers and mainframes. (This will become true of microcomputers also as 32-bit microprocessors become available.)

We are now involved in the third cycle. Microcomputers started out in the early 1970's as machines with very primitive instruction sets and small memories. Again, it was very important that programs be coded to require minimum amounts of time and space. As hardware technology has improved, microcomputers have become more sophisticated and their capabilities are approaching those of minicomputers.

Two distinct classes of microprocessors are emerging. The smaller, less powerful micros are suitable for replacing random logic and are not particularly well-suited to traditional computer applications. The faster,

more powerful microprocessors are suited for general purpose computing. Even among the latter class there are subclasses. There is quite a difference in the capabilities of the microprocessors commonly found in hobby computers and those used in more serious systems.

The instruction sets for some current microprocessors are identical to some minicomputers, since they are large scale integration (LSI) implementations of minicomputer architectures. Examples of such mini/micro-computer families are the PDP-11, Nova, TI990, Naked Mini and PDP-8. In some cases the I/O capabilities of the mini exceed those of the micro.

The newer microprocessors are more powerful, in some cases, than minicomputers of an older vintage. It is currently fashionable to compare instruction execution times for new microprocessors with those of the various models of the PDP-11. In addition, the newer microprocessors have address spaces which are larger than those of most of the older minicomputers.

Thus, the primary hardware differences between mini and microcomputers are packaging and I/O capability. Two other differences are the availability of software and the approach to marketing (selling components as opposed to systems). The dividing line between the two classes of machines is somewhat fuzzy. In a number of cases the CPU consists of a single integrated circuit on a chip (a portion of semiconductor material which comes from a wafer of the same material). Such a CPU is clearly a microprocessor, but a machine whose CPU requires multiple cards is clearly not. The gray area is for those CPUs which require a number of chips, but which fit comfortably on a card. It is not reasonable to restrict the definition of a microprocessor to a single chip CPU, since this would exclude such machines as the 8080 and F8, which are obviously microprocessors. (The 8080 requires two additional chips to supply the clock and bus interface logic). It seems appropriate to use a broad enough definition to include such machines as the LSI-11 which has a 2-5 chip CPU.

The definition which will be used here is to consider a CPU composed of 5 or fewer chips to be a microprocessor. This dividing line is somewhat arbitrary and could certainly be stretched, if necessary.

Another criterion which is sometimes used is that the CPU be composed of circuits which are sufficiently specialized so as not to be useful except as part of a CPU.

One important difference in these three cycles is the difference in the hardware costs, from one cycle to another. This is relevant particularly when considering the relative difference in cost between hardware and software. The cost of hardware has been decreasing, while the cost of software has been increasing. While hardware costs may not continue to decrease in absolute

terms, they will most likely not increase as fast as the cost of software.

Thus, while microprocessors represent the third hardware cycle, not all the conditions which existed in the early days of electronic computers prevail today. In addition to the disparity in software costs, additional knowledge has been gained regarding the use of development tools. It is possible to apply this knowledge to microprocessors, at least to the extent of using those tools which have been hosted on mainframes or minicomputers which were of comparable size to today's microcomputers.

Questions arise when attempts are made to apply the highest software technology to small machines. Certainly, a microprocessor cannot be expected to deliver identical performance of a mainframe. Is it possible, however, to provide acceptable performance for large-language compilers? What are the differences between a development system on a large host and one on a microprocessor? Is it possible to create a useful development system using a microprocessor as a host or would such a system merely be a toy? What are the advantages of a microprocessor-based system? The purpose of this study is to provide answers to such questions.

These questions apply to both Jovial J73- and Ada-based development systems, but particularly to Ada-based systems. Since Ada is a larger language, it places more requirements on the system. If a system can host an Ada compiler, it should also be able to host a J73 compiler. For this reason, and because Ada will eventually replace J73, the bulk of the effort expended in this study has been directed toward investigating the problems associated with hosting an Ada compiler and system on a microcomputer.

3. Microprocessor Technology

During the past ten years great advances have been made with integrated circuits, both with respect to circuit density and speed. Whereas, in the early 1970's microprocessors had primitive instruction sets and ran at clock speeds of about 200KHz [Noyce81], current microprocessors have instruction sets which are comparable to, or even better than, those of minicomputers of several years ago, and run at clock speeds of up to 15MHz [Nelson81].

There are a number of factors which are relevant to the problem of hosting large systems on a given hardware configuration. These include such factors as machine speed, addressing space, storage capacity, and I/O capability and speed.

Clearly, microcomputers are not as fast as larger machines. This is true for a number of reasons. One is the technology involved. Larger machines typically use Emitter Coupled Logic (ECL), whose gate delays are in the range 0.7 to 2 nanoseconds while most microprocessors use some form of Metallic Oxide Semiconductor (MOS) technology which exhibit delays of 4 to 200 nsec [Electronics, 4/17/80 p.547]. (For discussions of Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) see [Mead80, Clark80, and Tobias81].) Another factor which influences the processor speed is processor architecture. A number of large machines have multiple processors or multiple execution units. Instruction scheduling and various forms of pipelining may be used to increase throughput. Microprocessor architectures tend to be more straightforward, although some of the more recent microprocessors (e.g., Intel 8086, Zilog Z8000, Motorola MC68000) do use pipelining for instruction fetching. There is research being conducted into the use of multiple microprocessors; however, while this would be relevant to obtaining higher performance using micros, it is not pertinent to obtaining a minimum configuration for a development system. Another factor in the increased performance in larger systems is the use of cache memory. This feature is available for microprocessors, but it is relatively uncommon.

The speed of a machine also depends on the instruction set. Operations which are not present in the hardware (e.g., multiplication or division) must be simulated by the software. This can cause a considerable degradation in speed.

Early microprocessors, such as the 8080, had primitive instruction sets due to hardware limitations. Indexing had to be simulated by adding an offset to what would normally have been the contents of an index register. There were no instructions for multiplying and dividing and the only 16 bit operations were loading, adding and storing. Current microprocessors often provide operations such as multiplication and division for operands with lengths of 16 or 32 bits.

Index registers are particularly important if native code is to be generated for languages such as J73 or Ada which allow procedures to be recursive or reentrant. The lack of an index register would require that an address computation be performed for any memory reference to an item whose allocation was not static. The Z80 does have index registers, but lacks such instructions as multiply and divide.

The 16-bit microprocessors tend to have hardware to perform integer multiplication and division. Newer microprocessors such as the 8086, Z8000 and MC68000 can perform signed multiplication. However, most of these machines do not have floating point instructions built into the chips. The 8086 has a companion chip (the 8087) which does perform floating point arithmetic [Intel79]. Some of the look-alike microprocessors such as the Computer Automation Scout have floating-point hardware [Computer Automation]. (For a comparison of some of the 16-bit machines see [Toong81]. A survey of available microprocessors and microcomputers appears annually in EDN.)

Of the factors mentioned here, the most important in terms of the execution speed of system software is integer multiplication. While the presence of floating point hardware may be crucial to the performance of application programs, it is relatively unimportant with respect to system programs. Integer multiplication is most often used for subscript calculations in system programs.

3.1 Address Space

Until recently, addressing space was the most severe limitation which had to be dealt with. It is still an important consideration. Older microprocessors were limited to 64K bytes of memory or less. It is only recently that microprocessors which have the ability to address more than 64K at one time have been available. While some of the older machines had memory management units which would allow more than 64K of physical memory, the virtual address space for each user was limited to 16 bits, making it impractical to run programs larger than 64K. The 64K restriction has been recognized as the major limitation of the PDP-11 architecture [Bell.1978,p.38] and is also a limitation of most microprocessor architectures. The more recent machines such as the 8086, Z8000, and Motorola 68000 have a significantly larger address space.

There are two considerations concerning address space. The first is how much physical memory can be addressed by the machine. This is relevant to how many users can be accomodated before swapping is necessary. It is less of a factor if it is relatively easy to re-map memory.

A more important consideration with respect to hosting large programs is how much memory is instantaneously addressable or can easily be made addressable.

The 8086 has an address space of 1 megabyte but only 256K of this is instantaneously addressable since there are four segment registers and the maximum size of a segment is 64K. Since the size of the registers is 16 bits, the maximum size of a pointer or index (and hence a table or array) is 64K for all practical purposes. It would be possible to perform a double precision address computation and to load both a segment register and an index register, but this is undesirable, if it occurs often. The problem is less severe for subroutine references since a segment register could be loaded prior to a procedure call. (This would be analogous to loading a base register before calling a subroutine on the IBM 370). Thus, for all practical purposes there is a limitation of 192K (three segments of 64K each) for data on the 8086 but code is only limited by the size of memory (1M maximum addressable).

The addressing structure of the Z8000 is quite different. There are actually two versions of the CPU, one segmented and the other unsegmented. The unsegmented Z8002 CPU provides 64K byte addressability, but data references, stack references and program references can be separately distinguished, giving a total of 192K of addressable space in user mode. In system mode another 192K is addressable. Note that the full 192K can only be utilized if the data, stack and program spaces are each 64K. These figures are theoretical in that they represent the memory which can be addressed if some special pins are used by the support hardware. Not all Z8002-based microcomputers use them, so the actual amount of addressable memory may be

less.

The Z8001, the segmented version, allows up to 8M to be addressed in each address space, or a total of 24M for a given user. Each 8M is divided into 128 segments of 64K each. Since the index registers of the Z8000 only allow offsets of 16 bits (the index registers are double registers, but one half is used for the segment number), it is impractical to have tables or arrays larger than 64K. All 127 segments are instantaneously addressable, so this addressing scheme is somewhat more powerful than that used by the 8086.

Since the Digital Integrating Subsystem (DIS) machine is based on the Z8002 architecture, the remarks made above with respect to the Z8002 apply to the DIS. However, the actual configuration is 128K bytes, rather than the theoretical maximum of 192K. (The address space is divided into code and data spaces -- stack space is not separate.) Since the lower 2K bytes of data space are used for memory-mapped I/O and the lower 4K bytes of code space are used for the bootstrap loader, 122K bytes are actually available for use by programs [General Dynamics80].

The Motorola MC68000 permits 24M to be addressed, with the possibility of expanding the architecture at a future date to 4 gigabytes. The address space is unsegmented so the sizes of data structures are limited only by the size of memory (i.e., they are not restricted to 64K). The only difficulty with respect to address computation is that the maximum allowable displacements for various addressing modes are 32K or less, requiring that an extra instruction be generated in the event of a large offset. This is only a minor problem, however.

3.2 Auxiliary Storage

Auxiliary storage capacity is another important way that microcomputers differ from larger computer systems. A single disk pack for a 3330 type disk drive holds a hundred megabytes of data. Newer disks, such as the 3370, hold over half a gigabyte. Most microcomputer systems do not have that much total disk storage. (For a survey of microcomputer disk drives see [Roman81].)

3.3 Development Systems

There are already a number of development systems hosted on microprocessors [Santoni80, Mini-Micro Systems, August 1980]. They can be used to perform compilations or assemblies without requiring a minicomputer or mainframe host. It should be noted, however, that compilers residing on these systems are not for languages of the complexity of J73 or Ada. Compilers for subsets of COBOL, PL/I and Ada currently are hosted on microcomputers; these represent

the most complex languages now available on such systems.

Another facility supported by these development systems is In-Circuit Emulation (ICE). ICE allows the microprocessor in the target system to be replaced by a connection to the development system, so that the facilities of the development system can be used to check out the target system.

Some of these microcomputer based development systems support multiple users. The methods for doing this vary. Both centralized and distributed systems are used.

3.4 Minicomputer/Microcomputer Families

One of the reasons for using a microcomputer as the host for a development system is so that the application program under development can be run on a processor with the same architecture as the target. That is, the application can be tested on the development system without requiring a simulator for the target CPU. For larger projects, microcomputers may not provide sufficient throughput, particularly with regard to I/O.

One possible approach to meeting these requirements would be to use a minicomputer, from the same architectural family as the microprocessor target, as the host. There are a number of microcomputers which have the same instruction sets (or at least very nearly so) as some minicomputers. Some such families are the PDP-11s, Novas, Naked Minis, and the TI 990 and 9900.

Unfortunately, these microcomputer families suffer from the 64K addressing limitation.

3.5 Bundled Vs. Orthogonal Architectures

In an orthogonal architecture, a major function may be modified without affecting other major functions [Klingman, p.423]. By contrast, in a bundled architecture, modifying the capabilities for one function may have an effect on the capabilities of a number of others. The examples which Klingman uses for bundled and orthogonal architectures, respectively, are the Fairchild F-8 and the Intel 8080.

In the F-8 architecture, ROM, timers, I/O ports, interrupt logic and registers are all on a single chip. Thus, if ROM is added to the system, the timers, ports, etc. are also added. In the case of the 8080, adding ROM implies nothing more. I/O, for example, is unaffected.

While it is somewhat more awkward to expand a bundled architecture, such an

architecture does have the advantage of allowing low chip counts for the complete system. This is particularly advantageous for low-cost, high-volume applications. It is less important for a development system, since the CPU is a relatively small part of the total system cost. (The newer 16-bit microprocessors can be purchased for under $100 apiece in small quantities, but even a relatively small development system would cost thousands of dollars.) Of greater importance is the ability to be able to expand the system.

There are a number of single chip microcomputers available. Those currently in existence do not seem sufficiently powerful to serve as the host for an Ada compiler but such hardware should be available in several years. In some sense, though, a one-chip computer represents the ultimate in bundling.

The issue of bundled vs. orthogonal architectures is somewhat moot at this point, since those architectures at the high end of the microprocessor spectrum tend to be more orthogonal than not. It is primarily at the logic replacement end where bundling is likely to be an issue.

3.6 Current Microcomputer Systems

A microcomputer system, of course, is more than just a CPU. Although the power of the CPU must be considered when obtaining a microcomputer, it is necessary to address other hardware areas such as buses, memory and peripherals, as well as the overall configuration. In this section we attempt to provide examples of what hardware is currently available and how much it costs.

The systems listed here are only a small fraction of the number of different systems which are available.

Computer Manufacturers

There are a number of companies which produce microcomputer systems. These include the semiconductor manufacturers themselves, mainframe and minicomputer manufacturers, and companies whose involvement with computers is primarily as microcomputer builders. Semiconductor manufacturers who produce complete systems include Intel, Zilog, Motorola and Texas Instruments. Mainframe and minicomputer manufacturers whose offerings include microcomputer systems are IBM, Xerox, Hewlett-Packard, DEC and Data General. A large number of other companies produce microcomputer systems, but do not fall into either of the other categories. Some such as Apple are primarily computer manufacturers. Others, like Radio Shack are more diversified. Other manufacturers include Commodore, Atari, Ohio Scientific, Heath/Zenith and North Star. A discussion

of sales figures for the more popular makes can be found in [Infoworld, Sept.14,1981].

Buses

A bus is a common communication link between two or more system components [Bell78, Poe79]. External system buses are implemented as parallel lines on a printed circuit card. These lines can be divided into four major classes: address, data, control and miscellaneous (including clock, power, ground, and expansion) [Osborne80].

Most microcomputer systems use an external bus structure for communication between the CPU, memory and peripherals. There are exceptions, however. Single-chip microcomputers have memory on the same chip as the CPU, so the CPU can communicate directly with memory. Some single board micros communicate with peripherals via serial or parallel ports. A standard bus architecture (there are a number of popular buses) is better for adding peripherals and memory, since the appropriate boards (memory or controllers) may be plugged into the chassis. Although some peripherals can be added to a system via RS-232 interface, this sort of interface is typically slower than either that of a bus, or a parallel port. (RS-232 is a standard for connecting terminal equipment to communications equipment for serial data transmission at speeds of up to 20,000 bits/second [Ogden80]).

Some of the more popular microcomputer buses are the S-100 bus (IEEE-696), the IEEE-488, the Q-bus, and the Multibus. For each of these buses there are a number of peripheral controllers and memory boards available.

The type of bus is important from a performance standpoint as well as from the standpoint of peripheral availability. The bus structure may place limitations on the computer system in terms of both size and speed. Size limitations arise from the number of address lines available on the bus. Speed limitations arise if the bus cycle time is not sufficiently fast to satisfy CPU requests for data from memory. Some of the newer processors are too fast for some buses in the sense that the CPU must wait until data is available and therefore cannot run at full speed. One example of this is an 8Mhz MC68000 with a Q-bus. Since the bus cycle time is about a microsecond, the CPU speed is effectively halved when compared with that obtainable with a faster bus such as a Multibus.

Memory

Memory technology is advancing rapidly. RAM chips are increasingly common in the 64K size and 256K chips are expected to be available in the near future.

The major significance of this increase in density, with respect to microprocessor development systems, is that memory will become cheaper (on a per byte, rather than a per chip basis) so larger configurations will become more economical.

In order to take full advantage of the power of modern microprocessors, it is necessary for the memory to match the CPU speed. Different types of CPUs have different requirements. The 68000 requires that memory be only one-fourth as fast as the clock. The Z80 needs a memory with a cycle time equal to the clock cycle, and the 6502 requires that memory be twice as fast as the clock.

Memory management can be used to increase the amount of memory addressable by the CPU. This is particularly important when a CPU with a 64K addressing limit is being used to support multiple users, but it is also used in conjunction with some of the less restricted CPUs to allow contiguous logical address spaces to be mapped into fragments of real memory .

A number of techniques are used. Some systems use old-fashioned bank switching. This is a technique in which a bank of memory is activated or deactivated by a special command from the CPU.

Several semiconductor manufacturers have memory management unit chips available. These chips typically allow logical address spaces to be mapped into segments of physical memory [Johnson81Aug].

Disks

Progress is also being made in the technology of peripherals. With the advent of mini-Winchester disks, it was possible to attach low-cost, high-performance hard disks to a microcomputer system. There was a problem, however in providing backup for these disks, because the media were not removable. Floppy disks could be used for backup at the inconvenience of multiple media changes, because the capacity of floppies was considerably less than that of the Winchesters.

Streaming tape drives were introduced for the purpose of backing up these disks. The streaming drives differ from ordinary drives in that there is no inter-record gap; data is recorded in a continuous stream.

More recently, removable Winchester cartridges have been developed. A system with this sort of disk would not need additional hardware for backing up the database.

Systems

At the present time a large number of different microcomputer configurations are available. They range from programmable calculators (recall that they are as powerful as mainframes of the early fifties) to multi-megabyte systems. In addition, it is possible to configure networks of microcomputers which share a common database. The following section attempts to enumerate some configurations which are available. Not all are suitable for hosting an Ada development system, but they do illustrate what has been done and also provide a basis for predicting what will be done in the future.

At the low end of the scale are programmable calculators. These exist for an entirely different purpose than developing Ada programs. Next are the personal computers. These typically have small memories in their basic configurations and have limited input/output capabilities (often a cassette deck). Some examples of personal computers are the Apple II, the TRS-80 and the Commodore Pet. These computers exist in a different environment than development systems do, since the owner's time is usually cheap relative to hardware costs for a hobby computer, while programmers' time is expensive relative to the cost of MDS hardware. These systems are available in the neighborhood of $1000 (or less).

Larger 8-bit microcomputer configurations are applicable to more serious computing. These include more memory (around 64K) and more peripherals (at a minimum, floppy disks and a printer). There are a number of companies making systems which fall into this category, including: NORTH STAR, Cromemco, and Vector Graphic.

Note that the Apples, Commodores and TRS-80s can be configured in this category. Particularly noteworthy is the fact that for under $2000 it is possible to obtain a microcomputer with display, keyboard and floppy disk drives (Osborne I).

Systems based on 16-bit processors offer somewhat more computing power in most cases, although there is some overlap between the more powerful 8-bit machines and some of the less powerful 16-bit ones. Systems based on the 8086, Z8000 and 68000 are offered by the chip manufacturers (Intel, Zilog and Motorola) as well as by numerous other companies.

Western Digital has built the Pascal Microengine to execute P-code which was originally designed for interpretive execution. The Microengine uses the same chip set as the LSI-11, but with different microcode. The Pascal Microengine is being upgraded to be an Ada Microengine. A system with 128K bytes of memory and a single floppy disk drive sells for about $6900.

IBM has two microcomputer offerings. The smaller series is 8088-based and

sells for under $1600 in a minimum configuration. A larger version includes 64K, a floppy disk drive and a display for about $3000. The system is scheduled for delivery in October 1981 and is expandable to 256K bytes.

The other microcomputer series for IBM is the System/23, which is based on the Intel 8086. This system includes 64K RAM, 2.2 MB of diskette and a matrix printer for about $9400, and is already available.

Multi-User Systems

There are a number of different configurations for multi-user systems. Some of the smaller ones are based on 8-bit processors (typically Z80s). Multi-user configurations include traditional timesharing (one processor serves many users), and multi-processors with more than one user per processor. Even among the multi-processor setups there is disparity. Some are arranged as workstations on a network (similar to the Ethernet setup [Metcalfe76]); others have multiple processors in one chassis, providing a real machine for each user, where timesharing would only give the user a virtual machine.

One of the simpler and more inexpensive multi-user systems is the CT-80 system by Digiac. It is based on the Z80 and can handle up to four users in the largest configuration.

Action Computer Enterprise, Inc. has a Z80 based multiprocessor system with S-100 bus compatability. The system consists of a Z80 and 64K bytes for each user plus a service processor which handles shared resources. A two user system (with two user processors and a service processor) sells for under $6000. Additional user processors (Z80 + 64K bytes of memory) sell for about $1400. Peripherals are additional. A set of dual floppies costs about $2400, a 10M byte Winchester with controller about $3600, and a 26M byte Winchester around $4900. It is interesting to note that the CPU and main memory for a user costs less than even a set of floppy disks, and less than half that of a hard disk.

There are a number of Z8000 based multi-user computer systems available. One is Zilog's System 8000 which Zilog says has outperformed the PDP 11/70 in some benchmark tests [McCauley81]. This system is a traditional time-sharing configuration with a single Z8001 CPU. (Other micros are used for disk controllers.) Memory can be as large as 1M byte for the system. Unsegmented user programs can address up to 128K bytes (divided between code and data), and segmented programs can address up to 128 segments of 64K bytes each. The system uses Zilog's 8010 memory management units. The operating system on the System 8000 is Zeus, a modified version of UNIX. An eight-user system costs about $30,000.

Another Z8000-based system is the System X8000 from Computex. This system also uses the Z8001 as its CPU, but uses the Multibus. Total memory size may be up to 16 MB. The AMD (Advanced Micro Devices) 9511 floating point processor is available optionally.

The Computex system includes a memory mapping scheme which provides both segmentation and paging. Thirty-two segments are addressable without remapping. Sixteen different maps are available for sixteen different programs. The program number and segment number are used to compute a segment address and the segment address and the high 5 bits of the address are used to compute the page address.

This is an interesting memory mapping scheme because it allows users to share programs, but also to maintain separate address spaces without requiring a change of the map registers at a context switch.

ZMOS, the operating system which is available on the X8000, allows multiple users and multiple processors. It includes an Indexed Sequential Access Method (ISAM) file handler which allows multiple keys per file. There is also a database inquiry subsystem.

MC68000 Based Systems

In recent months a number of companies have brought out 68000 based microcomputers: Hewlett-Packard, Wicat, Computhink, Empirical Research Group, MicroDaSys, Forward Technology and Dual [Infoworld, Sept. 14, 1981].

Wicat systems has several 68000-based configurations. The System 150 is a workstation which is based on the Multibus and includes 256K bytes of RAM, a 5 1/4" floppy and a 10M byte Winchester with controllers for $8500. RAM may be expanded up to 1M byte.

The System 100 uses a proprietary bus and includes 256K bytes of RAM, a 20 MB Winchester with tape backup for about $21,500. Up to 6 MB of RAM can fit in the standard cabinet and the system can address a total of 14 MB.

One of the more interesting, though expensive, approaches to 68000-based systems has been taken by Apollo with their DOMAIN system. The DOMAIN is a network which is based on the workstation concept which was popularized by Xerox in their Ethernet. Each workstation has a bit map display, two 68000s, memory and peripherals. There are two classes of peripherals: Those which are integral to the Apollo system are connected to a block multiplexor channel. Other peripherals are connected through a Multibus.

Individual nodes may have from 256K to 1MB of main memory. One particularly

interesting feature of the nodes is that they provide multiple virtual terminals, so it is possible to monitor several independent processes at once.

The network itself is a ring structure which passes two-address packets at a rate of 10 M bits/second. There is a network virtual address space which allows one node to reference data which is stored at another node. This virtual address includes an object identifier and an address within the object.

A basic node (with 256K bytes) costs $24,000, and a mass storage expansion unit (with a 33 MB Winchester and 1 MB diskette) $10,000. Not all nodes in the network need to have peripherals, but at least one does.

Conclusions

At the present time, the 68000 seems to be the microprocessor of choice on which to base a development system, followed by the Z8001 and the 8086 (or 8088) in that order. Systems which only provide 64K addressability, including some of the older 16-bit architectures and virtually all of the 8 bit processors, are not large enough to support a production compiler for full Ada. Such systems could be very economical for subsets, however.

The Wicat 150 is notable in that it provides substantial capability for a reasonable price. The combination of the 68000 and the Multibus is particularily attractive.

[Franta80] provides a view of system selection.

3.7 Future Microprocessors

In the future it is virtually certain that significant advances will be made with respect to microprocessor capabilities. Some of the advances are not particularly relevant to this study. For instance, although decreased power consumption is important to hardware system designers and embedded computer applications, it does not have a direct effect on software development.

There are three expected developments which are very much relevant to software, though: increased processor speed, larger word size, and virtual memory.

Currently, the time required to perform a register to register addition is around 300ns for a fast MOS processor (e.g., an 8086 with a 10MHz clock). Bipolar technologies give somewhat greater speeds. Times of 20ns have been predicted for 1985 [Wise80].

Another major enhancement which seems imminent is the increase in word size to 32 bits. There are already 16-bit word machines which perform a limited amount of 32 bit arithmetic. Most of the newer 16-bit processors provide doubleword addition and subtraction. Several 32-bit machines have already been announced.

Intel has announced the iAPX 432 [Intel81, Hemenway81], a 32-bit machine with a System Implementation Language (SIL) which is purported to be a superset of Ada. The iAPX 432 is easily the most impressive microprocessor announced to date; in some respects it is more impressive than mainframes. Some early reports implied that the iAPX 432 executed Ada programs directly. This is not the case; programs must be compiled [Zeigler81]. However, the architecture is unconventional in a number of ways. It is an object based architecture. Protection is based on objects, rather than being on a user-by-user basis. No registers are visible to the programmer; operands come from memory or the stack. Instructions are bit-addressable; no alignment is necessary, nor are they required to be an integral number of bytes long. Multi-process and dynamic storage allocation mechanisms are present in the hardware. There is a very large (2**40 bytes) virtual address space.

Intel has also announced three more conventional microprocessors: the iAPX 186, the iAPX 188 and the iAPX 286, which will be upward compatible with the iAPX 86 (more commonly called the 8086). The iAPX 186 will be faster than the 8086 and will have some high-level language enhancements in the instruction set. The iAPX 188 has an 8-bit path to memory.

The iAPX 286 offers virtual memory and hardware support for such functions as context switches and operating system calls. Addressable memory is 16MB (physical) and 1 gigabyte (virtual).

National Semiconductor [National Semiconductor81] has also announced a 32-bit machine, the 16032. This machine will have a facility for implementing virtual memory. It will provide 16MB addressability directly or 32MB with a Memory Management Unit (MMU). There will be facilities for handling floating point numbers, arrays and strings of bits. A hardware breakpoint facility will be provided. Another novel feature is the use of a module map for external references so that if one module is changed (in ROM), other modules need not be recompiled or relinked.

National Semiconductor has also announced the 16008 and 16016, which are less powerful than the 16032, but are of interest because they will have two instruction sets, their own and that of the 8080. Through the use of escape operators it will be possible to switch back and forth between the two instruction sets.

Zilog has announced the Z8003, which is essentially a Z8001 with virtual

memory capability [Electronics, June 30,1981], and is working on a 32-bit micro.

Other companies which are said to be developing 32-bit microprocessors are ATT, IBM, HP, MOS Technology [Electronics, April 21, 1981, p. 124], and Nippon Telegraph and Telephone [Electronics, June 2, 1981.

Fujitsu [Inui81] and Toshiba [EDN, February 4, 1981] have announced 16-bit microprocessors, each of which is capable of addressing 16MB.

Mikros has announced a microprocessor version of the MIL-STD-1750A architecture [Electronics, July 28, 1981]. The implementation is Silicon-on-Sapphire (SOS) and will execute 200,000 instructions per second (.2 MIPS).

4. MICROCOMPUTER SYSTEM ADVANTAGES

There are a number ot advantages to using a microcomputer as a development system. These include:

1. Lower hardware cost.
2. More localized control of the computer system.
3. Guaranteed response.
4. Better security.
5. Fewer communication problems.
6. High bandwidth.
7. More reliable transfer of media.

4.1 Cost

A microcomputer system with a floppy-disk can be purchased for about $6,000. It is possible to spend that much for time-sharing for a few users in just a matter of weeks. Even with a hard disk it is possible to buy a microcomputer system for less than $10,000, although some of the larger systems exceed this figure.

The fact that hardware costs are so much cheaper for microprocessors opens up additional possibilities for configurations which would not be possible with more expensive hardware. It becomes possible to give each user his own CPU, because the economics have changed. Previously, it was unthinkable for the CPU to be idle, since the cost of the computer was so great. Now, it costs more for a programmer to be idle waiting for the machine, than it does for a machine to be idle waiting for the programmer.

Another effect that microcomputers have had on costs is with respect to software. Since the hardware is so inexpensive, more computers are sold, so there is a bigger market for software. Thus, it is possible for software producers to take advantage of the economies of scale and to offer individual copies of software at a fraction of the development cost, making up the difference in volume.

4.2 Localized Control

The fact that a microcomputer system is too small to be used by a large number of programmers means that the control over the machine is likely to be at a lower level than for a larger machine. There is less likely to be a conflict between different groups for the use of computer resources.

4.3 Response Time

The response time for a single-user system is consistent, since there are no other users contending for system resources. Consistent response has been shown to be an asset, even if the response is not particularly fast [Miller77]. Although the performance of a microprocessor-based workstation would not be the equal of a lightly loaded mainframe, it would be significantly better than that of an overloaded one.

4.4 Security

It should be noted that the use of the word "security" here refers to the prevention of unauthorized access to the machine, rather than the protection of the operating system from the users or the users from each other.

It would be easier to maintain the security of a microcomputer system than of a larger system because access could be restricted to fewer individuals. If a large machine is to serve a large number of users, all must have access to the system. It would be the responsibility of the operating system to prevent an unauthorized user from accessing data which is not supposed to be available to that user. The problem of unauthorized access is compounded if there are dial-up lines to the system. A microcomputer system can be physically isolated, thereby making unauthorized access more difficult.

Another aspect of security is the fact that it is not permissible to have classified and non-classified users on a single system at the same time. Since microcomputer systems can be smaller and cheaper, it is possible to have several complete systems for the cost of a larger system, allowing both classes of users to work simultaneously on physically distinct systems.

4.5 Communication

Since the microcomputer system will have fewer users than a large system, it is possible for them to be in close enough physical proximity to the machine for hardwired terminals to be used. This will eliminate problems of noisy phone lines and loss of carrier.

4.6 High-bandwidth

A related benefit of having hardwired terminals is that it is possible to obtain good performance from screen-oriented text editors, without requiring expensive communications equipment. These screen-oriented editors are both easier to use and less error-prone. They are easier to use because indicating where a change is to be made can be done by positioning the cursor. They are less error-prone because a change made to the text is reflected by the display on the screen, and because a larger context is visible to insure that the text

being changed is really the text which was intended to be changed.

Although high-bandwidth terminals are available for larger systems also, it is common for the larger systems to be served by dial-up lines to allow users spread over a wide area to access the computer. The fact that microcomputer systems are smaller and do not require special air conditioning, makes it easier for the machines to be near to the users.

4.7 Media Transfer

Paradoxically, there seems to be a greater problem transferring media between large computer systems than between microcomputers. It is not uncommon for a tape written at one installation to be unreadable at another. On the other hand, we have had no such problems with floppy disks.

5. MICROCOMPUTER SYSTEM DISADVANTAGES

In addition to the obvious limitations on microprocessor systems such as speeds and capacities, there are some more subtle differences between using a large timesharing system and a microprocessor-based system for development. A centralized system allows costs for services to be spread over a large number of users, where it would be impractical to provide the same services on a machine with a smaller number of users.

5.1 File backup

One such service is the automatic backing-up of files. Most large systems automatically (at least from the user's point of view) save disk files on tape. While the periodicity for this service varies from system to system, some such facility usually exists. On a micro-based system, file backup is normally the user's responsibility.

5.2 Maintenance

Maintenance is another service provided by personnel associated with large computer systems. For smaller systems, it would be impractical to provide such personnel. It is likely that the users of the system would bear a larger responsibility in this area also, at least to the extent that they would need to be able to determine that a hardware problem existed and to summon assistance.

5.3 Software Distribution

Another problem which would be more acute is that of the distribution of system software, common utilities and interface descriptions. The logistics of maintaining software on a large number of small systems would be more complicated than maintaining the software on a smaller number of large systems. It would be necessary for greater numbers of people to be in communication with the software maintainers since each copy of the software would serve fewer users. Additionally, any time there were revisions made to the software or interfaces, it would be necessary to install them on a greater number of systems than if the users were served by large systems.

5.4 Swapping/Paging

Large systems generally have special peripherals to speed up swapping or paging. The devices used may be drums, fixed-head disks, or, more recently,

semiconductor disk emulators. These devices eliminate seek time, greatly improving performance when I/O requests do not involve sequential accesses. Such devices are not yet economical for microcomputer systems, since the cost of such a peripheral would greatly exceed that of the rest of the system components.

5.5 Machine Simulation

One method of program development which would not be as effective on some micros is machine simulation. A simulator for the target machine can be written on the development system and programs which will be run on the target can be checked out on the development system. In addition to providing additional debugging facilities, such as instruction traces and breakpoints, simulators can be used so that software can be checked out before the target hardware is available.

While simulation is costly in terms of CPU time required, nonetheless, on a fast machine, the real-time performance is acceptable. On certain microprocessors, however, there would be a marked performance degradation as compared with direct execution on a target machine. For some applications which are not CPU-intensive, the performance decrease would be acceptable. There are a number of applications for which the UCSD P-code interpretation gives acceptable performance, and the P-code is at roughly the same level as the instruction sets of some of the more powerful microprocessors. For other applications, however, the decrease in performance is noticeable. One example is that of searching for a character string in a file which has already been read into main memory.

This is only a problem on the less powerful microprocessors, however. Some of the more powerful microprocessors (such as the more recent 16 bit machines) are faster than some larger machines, at least for operands which are no longer than 16 bits. For certain CPU-intensive jobs, it is possible to achieve better performance on the more powerful micros than on some older mainframes, such as the DEC-10 KA10 (see Appendix M).

If a good development system is hosted on a microcomputer, it is likely to be used for targets which differ from the host microprocessor. This would require some form of simulation, were the application to be checked out on the development system.

6. PROSPECTS FOR A MICROCOMPUTER-HOSTED DEVELOPMENT SYSTEM

The present state of microprocessor technology is such that it can safely be said that it not only will be possible to host a useful development system on the hardware which will be available in 1985, but that today's hardware is already sufficient. Although microcomputers are not well-suited for projects which require enormous databases, they are more than adequate for moderate-sized projects.

Most of the tools which are traditionally part of development systems should be hostable on a microcomputer with no particular problems. Such programs as editors, linkers, loaders, document formatters and file manipulation utilities exist on current microprocessors. While the existing programs do not always meet "production" standards for features (e.g., editors may be restricted to modifying files which can be held in memory or linkers may not support overlaying), they are sufficiently close that success at hosting "industrial strength" versions on a microcomputer may be safely predicted.

Most of the tools mentioned above can be hosted on a microcomputer without a redesign effort. There are other tools and parts of the Ada Programming Support Environment (APSE) [Stoneman80], however, which may be restricted in scope or require a redesign if they are to be hosted on a smaller machine. These include the operating system itself, the debugger, the configuration manager and the Ada compiler.

The operating system on a microcomputer will be less elaborate than that of a larger machine. The original KL10 Monitor for the DEC-10 required about 90K 36-bit words (which translates roughly to 405K bytes) [Bell, 1978, p.506]. This is too large to be economical for a present-day microcomputer. (It could be done but it would result in a system far larger than a minimal system. It is interesting to note that the original PDP-6 Monitor took only 6K 36-bit words, or about 27K bytes [Ibid]).

In Stoneman [Stoneman80] the possibility of an APSE running on a underlying host operating system is discussed. It may not be possible to afford the overhead of having an APSE running on a host operating system on a microcomputer. Thus, it may be necessary to have a KAPSE which runs on a bare machine.

6.1 Compiler

The compiler is the largest tool in the system, and is the most important single factor, other than the application itself, in determining how big the microcomputer need be. J73 and Ada are both large languages; Ada is the larger of the two. Since these languages are large, their compilers are

necessarily larger than those for simpler languages such as Pascal. There are available a number of low-cost microcomputers capable of hosting Pascal compilers. However, many of these systems are limited to 64K bytes per user because the CPU uses 16 bits for addresses. It is doubtful that an Ada compiler could be hosted on such a machine without suffering an unacceptable performance degradation. A compiler for a subset of Ada, however, could be hosted on that type of system. Because the statement of work forbade the consideration of subsets, performance estimates in this report are based on the assumption that the compiler implements the entire Ada language. It must be remembered that hardware costs would be lower for a system capable of hosting a subset of the language than for one which could host the entire language.

6.2 Debugger

It is certainly possible to put a debugger on a microcomputer. Indeed, there are several debuggers which run under CP/M and which allow memory contents to be displayed and modified and breakpoints to be set. However, if a source-level debugger is desired, there may be some problems, even on a larger machine.

In order for the debugger to be at all useful, it must be able to access data which, according to Ada scope rules, should not be visible to it.

The optimizer phase of the compiler will be in conflict with the debugger, since code optimization tends to obscure the relationship between the source and object. Optimizations such as dead code elimination, code motion, global register allocation, folding and dead store suppression may cause confusion due to an apparent discrepancy between the source and the code executed. While these optimizations give the correct results with respect to the final output, the intermediate program states may appear incorrect.

Consider the following program fragment:

```
I:=J;      [1]
K:=I;      [2]
I:=L;      [3]
```

An optimizer may treat the program as if it had been written:

```
K:=J;      [4]
I:=L;      [5]
```

If the user requests that the value of I be printed after the execution of [1], the value or I may bear no relation to J due to the fact that J has been

folded into [2] giving [4], and [1] has been suppressed.

Another problem which appears to be more serious is that of debugging multiple tasks. Note that this problem is peculiar to Ada and does not apply to J73, which does not provide tasking facilities.

A problem which is more acute on a microcomputer is that of space, both for the debugger while it is executing and for the debugger tables on disk. If the target CPU and the development system CPU have the same amount of memory, and the application program's size is near to that of the target, there will not be enough room for both the debugger and the application. The internal symbol dictionaries (ISD's) for the application are potentially quite large. This could be a problem on a system which didn't have much disk storage available.

6.3 Configuration Management

On a large system, an elaborate configuration management scheme may be necessary to keep a project from getting out of control. It is necessary to be somewhat more conservative on a microcomputer-based system. The facilities provided by a number of popular microcomputer operating systems are rather limited in this regard although some do provide date/time stamps and the ability to mark files as being read-only. Additional facilities are required for a serious development system to make it easy to maintain distinct versions and revisions of programs.

7. COMPILERS

The compiler is the tool which is likely to be the most difficult to host on a small machine. It is here that a microprocessor's speed and space limitations become acute. There are a number of compilers available on microcomputer systems. The languages they support include Pascal [Jensen74, JRT80, Sorcim79, UCSD78, Intersystems80, and Microsoft80], BASIC, subsets of FORTRAN and COBOL and, more recently, subsets of PL/I [Digital Research80] and Ada.

Since this study began, three companies (Telesoft, Western Digital and Digital Electronic Systems) have announced Ada compilers which are hosted on microcomputers. At the present time these compilers are incomplete implementations of the language. A fourth (RR Software) has announced a JANUS compiler which is said to compile a subset of Ada.

The languages that these compilers support are substantially smaller than either J73 or Ada. In addition, the compilers lack a number of features that are normally found in production compilers.

Pascal compilers were examined during the course of this study for several reasons. First, there were a variety of different implementations to choose from (compiled and interpretive). Second, except for Ada itself, of the microcomputer languages, Pascal is the one which most resembles Ada.

It should be noted that the Ada compilers were not available at the time the Pascal compilers were being studied.

7.1 Language Features

It is useful to compare language features to obtain some idea of how much different a compiler for J73 or Ada would be than one for Pascal. The following is a list of some of the major differences between J73 and Pascal.

Features which J73 has but Pascal does not:

1. Source macros (defines)
2. External procedures (although some implementations of Pascal allow them)
3. Compools
4. Character string variables (although some implementations allow them)
5. Specified tables
6. Parallel tables
7. Tables with multiple entries/word (tight tables)
8. Tables with variable dimensions (star tables)

9. Bit and byte functions
10. Status lists (with user specifiable values)
11. Overlays (for allocation of data)
12. Presets
13. Inline procedures
14. Reentrant procedures
15. Conditional compilation
16. User-specified procedure linkages
17. Fixed-point arithmetic
18. Compile-time arithmetic

Features of Pascal which are not in J73 include:

1. Sets (although bit strings can be used to simulate them)
2. Subranges/range checking
3. Input/Output
4. Variant records (although specified and LIKE tables can be used to simulate them)
5. Records of arrays

The following lists enumerate some of the differences between Pascal and Ada.

Features of Ada which are not in Pascal include:

1. Separate Procedures
2. Packages
3. Generics (user-defined)
4. Overloading
5. Aggregates
6. Slices
7. Tasking
8. Exceptions
9. Enumeration types with user-specified values
10. Arrays with dynamic bounds
11. Fixed-point arithmetic
12. Representation specifications for records
13. Inline procedures
14. Array Catenation

Features of Pascal not in Ada include:

1. Sets

A discussion of the translation between Pascal and Ada can be found in [Albrecht80].

7.2 Comparison of J73 and Pascal Compilers

Two microprocessor Pascal compilers were selected for comparison: UCSD Pascal [UCSD78] and Pascal/Z [Intersystems80]. They were chosen because they represent two different approaches. UCSD Pascal generates P-code which is interpreted except when it is run on a Pascal Microengine .Western Digital]. Pascal/Z was selected over other low-priced compilers because it generated native code, and, at the time it was obtained, it was the only one which implemented virtually the full language.

In addition to the fact that Pascal is a smaller language, there are a number of other differences between the existing J73 compilers and microprocessor Pascals. These include:

1. Quality of code produced
2. Compiler listing options
3. Compiler outputs

The level of optimization in the SEA J73 is higher than that found in the microprocessor Pascals (USCD and PASCAL/Z) which were examined in this study. The UCSD compiler generates straightforward code and PASCAL/Z does some local optimization, but neither does any regional analysis.

Production compilers normally generate cross reference and attribute listings, define expansions, and assembly listings. Microprocessor Pascals typically do not provide such listings. It is possible to get an assembly listing from PASCAL/Z, but only because the compiler generates macro code which is then run through an assembler. Other listings such as reformatted source and compiler statistics (e.g., types of statements, tokens, etc) are not available from the Pascals.

Compiler outputs such as reformatted source files, and compool files, are not available from the Pascal compilers. Relocatable object modules are not generally available. In the case of Pascal/Z, the relocatable is output by the assembler rather than the compiler.

Although it is a PL/I rather than a Pascal compiler, PL/I-80 does possess a number of these features. It does generate relocatable object modules, assembly listings and attribute listings. It is, however, a multi-pass compiler [Infoworld, Sept. 1, 1980].

The microcomputer Pascals, in general, run in 64K bytes or less. The two mentioned above both run on a 56K (byte) system, although each is rather cramped in such a region. In order to compile a 400 line benchmark program through Pascal/Z, it was necessary to break up the benchmark. The UCSD Pascal compiler could compile the entire benchmark in swapping mode, but was unable

to compile one of the pieces with swapping turned off. In contrast, the individual phases of the SEA J73 compiler are substantially larger. The largest phase in the DEC-10 implementation requires 61,040 words (roughly the equivalent of 274,680 bytes), while the IBM 3033 version takes 81,428 words or 325,712 bytes (see Appendix A).

The difference in compiler sizes is apparent on disk also. Pascal/Z takes 56K. The UCSD Pascal compiler takes 35K. The SEA compiler on the DEC-10 takes about 1.2 megabytes (see Appendix B).

In order to get a better feel for the problems which would be faced in trying to achieve acceptable compiler performance on a microprocessor, compiler statistics were gathered and some benchmark programs were created. There are three conflicting goals in hosting a compiler on a microprocessor: making the compiler small, making it fast, and generating efficient code. While the first two goals also exist on large systems, they are not as critical there. The third is more a function of the target than of the host, but it compounds the problems of trying to meet either of the other goals.

The size of the compiler is dependent on several factors including design, implementation technique and code expansion. The design is the single most important factor. Since the SEA and JOCIT compilers were designed to run on large systems, it was possible to have only 5 passes, in order to minimize the amount of I/O to temporary files for communication between phases. Another design decision which sacrificed smallness for the sake of speed in the case of the SEA compiler, was to have a resident symbol table. A different design would have made it possible to ease the compiler's space requirements, at the cost of a decrease in speed.

Implementation techniques which can be used to keep compilers small include the use of interpreters and reusing one-time initialization code as buffers. Interpreters can come into play at several levels. The compiler source may be translated to an interpretable form (such as P-code) and interpreted. This is the technique used in the UCSD Pascal compiler. Another possibility is to write an interpreter as part of the compiler and have parts of one or more phases table driven.

If the compiler is written in a high-level language, then its size will be determined, in part, by the type and quality of code generated by the compiler through which the compiler itself is compiled. As mentioned above, one possible alternative is to generate interpretive code. Even so, there can be differences in the amount of code generated per line of source. Native code can be expected to take more room than interpretive code. Threaded code [Bell73] is a compromise between the two, providing faster execution than interpretive code, but requiring less room than native code.

7.3 Microprocessor Ada Compilers

Telesoft

The Telesoft Ada compiler is hosted on 68000-based microcomputers and generates native code for the 68000. The compiler is coded in Pascal and is composed of three phases. The first phase is the front end, the second translates from one form of IL to another, and the third is the code generator. The compiler operates on a 256K (byte) system. About 300KB disk space is required for the compiler.

Although the implementation is not yet complete, a healthy subset of the language has been implemented including: packages, some tasking, and exceptions (except for anonymous blocks). Still to be implemented are: generics, representation specifications, complex overloading, and those aspects of tasking which involve the clock.

We had the opportunity to run some benchmarks through this compiler. (The results are shown in Appendix J.) At the time the compiler was hosted on an 8MHz 68000 with a Q-bus and a Winchester disk. For one rather informal test compile speed was slightly over 250 lines/minute (wall clock). This particular compiler was compiled with range checking turned on; with checking turned off it would be faster and smaller. (The J73 compiler statistics are for compilers without range checking; thus, comparisons of compiler speed between J73 and Ada would be unfair without taking this into account.) It should also be noted that the Q-bus is slow relative to the 68000, and does not allow the CPU to run at full speed. We have been informed by Telesoft that the compiler has been hosted on a 68000 with on-board memory and now runs at over 550 lines/minute.

Western Digital

Western Digital has an Ada compiler which runs on their Microengine. It also is an incomplete implementation at this stage.

Digital Electronic Systems

Digital Electronic Systems has advertised an Ada compiler for a variety of hosts and targets. They plan a phased implementation with a goal of compiler certification. We have been unable to obtain any hard information about this system.

RR Software

RR Software advertises a Janus compiler, which is for a subset of Ada. It appears that the subset currently implemented is roughly equivalent to Pascal with the addition of packages.

7.4 Characteristics of J73 Code

The statistics collection feature of the SEA J73/I compiler was used to obtain information about the types of statements and operations which actually appear in the J73 compiler. (Note that the J73 compiler studied was implemented in J73/I). It should be noted that these statistics obtained are for a particular compiler design and implementation. One aspect of the compiler design which has a great impact on the results obtained is the fact that the symbol table is resident and is treated as an array. A compiler which has a non-resident symbol table, or which uses pointers rather than subscripts as symbol table pointers would exhibit different characteristics.

The statistics which have been compiled are shown in Appendices H and I.

A number of observations can be made based on these figures. The first is that a relatively small number of token types account for the majority of the tokens which appear in the program. This implies that if efficient code (in terms of space) is generated for these types of tokens, that the overall code will be small. Conversely, optimizations performed for most operators will actually have little effect on program size.

The token which was the most common by far was PRIM, which represents an operand. Two other common tokens are SUBS and AT which are used for subscription and pointer dereferencing. For the IBM 360 compiler MUL, (multiplication) was also common. On the DEC-10 multiplication was far less common. On the 360 a multiplication by 4 is inserted for each subscript operation because the 360 has four bytes per word. The code generator and optimizer change these multiplications to shifts or include them in constant offsets.

Other tokens which occurred often were STNO (statement number) and ATTR (set the attributes of a operand). Neither of these operators causes code to be generated. The fact that STNO is common indicates that if debugging code were to be inserted at each statement, the effect on the size of the object program would be great. Another implication of the relative frequency of STNO operators is that the average number of operators and operands per statement is small.

Two other operators which were quite common were REPL and PARM. REPL

(replace) is the assignment operator and PARM is used for passing parameters.

Thus, of the common tokens for which code is actually generated (this excludes STNO, ATTR and LABL (label definition)), almost all are concerned with addressing operands rather than performing operations on them. If subscript multiplications are excluded, the most common arithmetic operators are comparisons, which occur far less frequently than any of the tokens discussed here.

Thus, it can be seen that operand addressing is an area where there is potential for a high payoff for optimization. A number of machines have short and long forms for addresses. Making good use of such machine features is important in reducing the size of object programs.

Note that the statistics discussed here are static. Thus, it is not possible to draw any conclusion as to fruitful areas for execution speed optimizations from these statistics.

7.5 Compiler Code Expansion

The statistics obtained were used to select three procedures which collectively reflect the composition of statements and operators found in the compiler. Actual procedures from the compiler were used in order to better reflect the usage of language constructs -- in particular, the interaction between statements (e.g., for common subexpression elimination).

The benchmark programs were compiled on the DEC-10 using the SEA compiler and translated into Pascal to be compiled using both the UCSD and Pascal/Z compilers. The translated versions followed the original J73 as closely as possible. Some modifications had to be made to allow for the fact that Pascal does not have external procedures (although the UCSD implementation permits them), a bit substring operator, nor a return statement. The external procedure references were handled by including dummy internal procedures. The code for these dummy procedures was excluded from the totals. BIT functions which selected a single fixed bit were replaced by references to boolean variables. There was one use of a bit string with a variable first bit which was translated to a set membership operation. Return statements were translated as gotos to labels at the end of the appropriate procedures.

Figures obtained during this study (see Appendix G) indicate that the sizes (in bits) of the code generated by the DEC-10 compiler and that generated by the Pascal/Z compiler are approximately the same, while the P-code generated by the UCSD compiler is approximately 60% smaller. No allowance has been made for the size of library routines required by the J73 and Pascal/Z versions or the size of the UCSD interpreter. In addition, the data in J73 were

statically allocated (in the J73 rather than Pascal sense of the phrase) while the data for the Pascal compilations were allocated on the run-time stack. The sizes of the procs are only for the code, but there would have been some differences in the code generated for the J73 compilation, if the data had been dynamic. Table references, in particular, would have required additional code on the DEC-10.

If, indeed, the 60% size reduction figure held for the entire compiler, then the SEA compiler would require about 110K bytes, exclusive of symbol table space and room for the operating system and P-code interpreter. This would allow only about 18K to be used for the system, the interpreter and the symbol table on a 128K machine.

It would, however, be difficult to achieve this size reduction because the P-code machine upon which the figures are based, handles integers and addresses which are a maximum of 16 bits long. A provision would have to be made to allow addresses which are longer. This will have an impact both on the size of the interpreter and on the size of the code. In addition, only small reductions are possible in the size of the data, since most of the significant tables in the compiler are both packed and global. (The symbol table is the prime example of such a table.)

7.6 Compilation Speeds

As was mentioned previously, speed of compilation is another important consideration. Unless programs can be compiled at a speed of several hundred lines per minute, the delay when a large module is being compiled will be so great as to turn the system into what amounts to a batch system with interactive editing.

Some timing statistics were obtained using the SEA J73/I compiler to get a better picture as to what sort of compilation speeds can be expected on a mainframe. There are several speeds which are relevant to the useability of a compiler. The most important from the user's point of view is the wall clock time since that is the measure of when the compiler's output is ready to be used. The speed of the compiler with respect to lines per CPU minute is less important with respect to programmer productivity, although it can exert an influence both on the wall-clock time required and system throughput.

The performance figures for compilations of six different modules, ranging from just under 3000 to almost 6000 lines are available (see Appendix C). The compilations were performed using the SEA J73/I compiler on a DEC-10 KL10 under the TENEX operating system. Real-time compilation speeds ranged from 235 lines/minute to 2300 lines/minute. It is interesting to note that this 10:1 difference in performance was obtained for two compilations of the same

module. This difference is due entirely to the load on the system. Lines/CPU minute figures were in the range 4200 to 6800.

These figures were obtained for compilations with the listing options turned off. In particular, no cross reference listing was requested. In another run not reported above, a cross reference listing was obtained. The CPU time required increased by one-third, and although the lines/CPU minute figure was lower than any of the others by over 1000 lines/minute, the wall-clock lines/minute fell in the middle of the group.

Appendix D contains figures obtained on a KI10 running the TOPS-10 operating system. The perceived speeds are in the same range, although the lines/CPU minute figures are lower due to the slower CPU.

A useful microprocessor development system does not need even to approach the CPU speed shown by the compiler on the DEC-10 since the DEC-10 user must share the CPU with numerous other users. While the microprocessor is less powerful than a mainframe, there would, presumably, be fewer users on the microprocessor, so it should be possible to achieve reasonable performance by limiting the number of users.

Compilation speeds were obtained for two different Pascal compilers on a North Star Horizon. The Horizon is based on a Z80 microprocessor with a clock speed of 4 MHZ. Peripherals on the system on which the tests were run include dual 5 1/4" floppy disk drives, a CRT and a printer. The disk has a transfer rate of 250K bits/second, an average latency of 100 ms and an average access time of 463 ms. [Shugart77]. The disks are formatted with 10 sectors/track and 35 tracks/disk. The disks are double density with 512 bytes/sector and 175K bytes/disk. One of the most important facts to be noted about this system is that it uses programmed I/O rather than interrupt-driven Direct Memory Access (DMA). This can have a significant effect on speed of execution, since the CPU is effectively idle whenever I/O is being performed.

In contrast, a cartridge disk such as the RL01 holds over 5 megabytes, has a transfer rate of 512K bytes/second, an average latency of 12.5 ms and a seek time of 55 ms [Dec80]. Winchester disks have comparable specifications.

The two Pascal compilers on which the test cases were run are the UCSD Pascal and the Pascal/Z compiler. The UCSD Pascal compiler is itself written in Pascal. Its execution is interpretive, since it has been compiled to P-code rather than native code. The P-code contains several operations which are intended specifically to enhance compiler performance. These include the treesearch and idsearch operators [UCSD78].

The compiler is one pass, but may be operated in swapping and non-swapping modes. Swapping mode causes slower execution, but allows larger programs to be

compiled. Since the symbol table entries for a procedure's local data are purged at the end of the procedure, a program which is "too large" may be shorter than another which can be compiled.

The Pascal/Z compiler generates assembly code containing both machine instructions and macro calls. The output of the compiler is assembled with a set of macro definitions to produce a relocatable; the relocatables are then linked with the library to create an executable module.

The compiler itself is one pass, although it could be argued that the assembler is, in effect, the second and third passes of the compiler. Indeed, some of the compiler options are handled in the macro expansion. Options such as turning range checking on or off are implemented by generating an assembler pseudo-op indicating whether this option is being turned on or off and letting the range check macro determine whether or not to generate code based on the value of the flag.

Timings were obtained for the three benchmarks discussed above as well as for several other programs. Due to incompatibilities in the languages processed (particularly with regard to strings), not all programs were compiled using both compilers. Timings were made using a stopwatch. The time required for human interaction was excluded. (The human interaction consisted in the case of UCSD Pascal of designating the source and object files; in the case of Pascal/Z it was typing the command line).

Runs were made with and without listings for both compilers. When source listings were requested they were written to the disk. The same disk drive was used both for source and object files.

The highest speeds obtained were with the UCSD Pascal compiler, compiling without listings and without swapping. Speeds ranged from 275 lines/minute to a high of just over 380 lines/minute. With listings off but swapping on, speeds were in the neighborhood of 275 lines/minute. The worst results were obtained with listings and swapping on. For these latter compilations speeds ranged between 70 and 100 lines/minute.

The figures were somewhat different for Pascal/Z. For compilation only and with listings turned off, speeds were in the range of 110 to 130 lines/minute. With listings turned on, speeds declined to the 90 to 105 lines/minute range. The speeds for compiling and assembling were roughly half of the speeds for compilation alone. Alternatively, the assembly times were comparable to compilation times. For compiling and assembling with no listings speeds in the range of 55 to 70 lines/minute were observed. With source listings the range was from 50 to 65 lines/minute. When assembly listings were produced the range was 30 to 40 lines/minute.

One of the more interesting results was the fact that obtaining a source listing has a great deal of effect on the UCSD Pascal compiler but does not really have much effect on Pascal/Z. One possible explanation is that the size of the source listing file is large compared to the UCSD object file but small in relation to the Pascal/Z assembler input file. Another is the fact that there is a high (1 second) cost associated with starting up the disk drive if it is shut down due to inactivity. It is, in fact, possible to envision a case where it takes less time to do more I/O, due to the fact that the process of restarting the motor is so time-consuming. If the motor were kept running due to the higher volume of I/O, less time might actually be required.

These compilation speed figures indicate that on a hardware configuration like that of the Northstar it is difficult to obtain sufficiently high compilation speeds so that J73 or Ada compilers could be used for serious development work. The speeds for the UCSD Pascal compiler with listings turned off are well within the acceptable range. It must be remembered, though, that it would not be possible to fit a one phase J73 or Ada compiler in such a small machine. Thus, there would be additional overhead due to overlaying and the necessity of saving information on disk between overlays.

The use of interrupt-driven Direct Memory Access (DMA) for I/O is likely to provide a significant improvement in speed. Hard disks (Winchester or Cartridge) could also help to raise compilation speeds significantly.

Compilation speed estimates for various hardware configurations appear in Chapter 20. Figures for machine speeds for an instruction mix like that which is found in compilers are given in Appendix M.

7.7 Execution Speeds

A number of execution speed tests were run in order to check the difference in speed between compiled, threaded and interpreted code. These tests were conducted on the Northstar Horizon described above. The compilers used were Pascal/Z for compiled code and UCSD Pascal for interpreted code. Threaded code was obtained through the use of a Forth system [Supersoft80]. The test cases were hand-compiled into a dialect of Forth, and threaded code was obtained by compiling these definitions. It should be noted that Forth was used strictly as a means for getting threaded code, not for the purpose of comparing Pascal and Forth.

Several different types of tests were run: prime number generation, Ackerman's function, and sorts. Of these tests, the sorts are probably the most similar to the type of code found in compilers. The other tests were used to compare various other aspects of program execution.

Two different prime number generators were run. One used the sieve of Eratosthenes and the other used division. For the sieve, the compiled code executed slightly less than six times faster (two times faster with range checking), but printing the results took almost three times longer. (This is due in part to the fact that UCSD Pascal prints integers left-justified, but Pascal/Z inserts leading blanks.) When division was used to compute the prime numbers, the interpreted code was nearly three times as fast as the compiled code. These tests amounted to a test of the interpreter against the runtime library for the compiler, and are not particularly valuable in predicting performance.

The Ackerman's function test is a good indication of the relative costs of subroutine calls for the two systems. In this test the compiled code was almost three times as fast.

Several sort tests were run. Recursive and iterative versions of quicksort as well as bubblesort were tested. The programs were obtained from [Wirth76]. Interestingly enough, the recursive quicksort ran faster than the iterative version in both compiled and interpretive code. Bubblesort was slower, even for small values. The interpreted code was almost exactly twice as slow for all three versions of the sorts.

As an additional test, bubblesort and the recursive quicksort were run with bounds checking turned off. These two sorts were also run as threaded code. The difference in speeds between the compiled and interpreted code rose from 2:1 to 5:1. The threaded code was the slowest with a ratio of 6:1 to the compiled code. The reason for this is that the Z80 is not well-suited to interpreted code (there is no autoincrement addressing mode) and because the P-machine is at a higher level than the threaded code. (This is particularly true with respect to address calculations.)

Range checking had a drastic effect on execution speed. The compiled code executed almost 3 times slower with checking turned on. Interpreted code ran only about 1.2 times as slow. The fact that checking had more of an effect on the native code is not at all surprising since the cost of checking is roughly the same in absolute terms for both compiled and interpreted code, but since the interpreted code is slower, checking is cheaper in relative terms.

Several of the tests were also run through the Telesoft Ada compiler on the 68000-based system described above. These tests are not a true indication of the difference in power between the Z80 and the 68000 because the 68000 was restricted by the Q bus, but the tests do provide a lower bound on the performance improvement which could be expected from compute-bound jobs. Speeds on the 68000 were from slightly less than 2 to slightly more than 4 times faster than on the Z80.

In order to obtain some baseline figures for comparison, some of the tests were run on various models of DEC-10s. Of greatest interest for the purposes of this study is the comparison between the DEC-10 speeds and the 68000 speeds. In several cases the speeds achieved on the 68000 were faster than those obtained on the KA10. The test cases were compiled using the Telesoft Ada and Pascal compilers on the 68000, and using a compiler developed at the University of Hamburg on the DEC-10. The quality of object code generated for the two machines was comparable when bounds checking was turned off, so the test cases without bounds checking are a good indication of relative machine speeds. None of the compilers performed any extensive optimization. The Telesoft Pascal compiler made better use of machine operations for statements such as "I := I + 1;" than did the DEC-10 compiler. The Telesoft compiler generated reentrant code, whereas the DEC-10 compiler allocated program-level data statically. The quicksort test case was run with the array statically and dynamically allocated on the DEC-10, in order to provide a better picture of the relative machine speeds. In the case that the array was allocated statically, the DEC-10 code for array references is simpler. When the array was moved to an inner block, it was more complex. Code equivalent to the Telesoft Pascal generated code would fall somewhere in between.

Range checking slowed the 68000 code significantly more than the DEC-10 code, because checking was done in-line on the DEC-10 (with two skip instructions), but by means of a subroutine call on the 68000.

Other benchmark results have been published in [Gilbreath81], [Grappel81], [Titus81], and [Interface Age, June 1980]. One unfortunate aspect of benchmarking microprocessors is that the results are out of date almost as soon as they are obtained, since manufacturers are often bringing out faster versions of the various chips.

[Grappel81] compared four 16-bit micros: the LSI-11, the 8086, the Z8000 and the 68000. In the tests as conducted the 68000 fared best overall, but Zilog recently has run ads [Electronics, Sept. 7, 1981] in which the figures were recalculated for a 10 Mhz Z8000. (The original tests used a 6 Mhz Z8000.) Not surprisingly, the Z8000 was the best with the revised figures.

Gilbreath [Gilbreath81] ran an Eratosthenes sieve algorithm through a number of compilers on a large variety of microprocessors. Unfortuntely, his results are not directly comparable with ours since the alogrithms differed. The 16-bit processors, in general, did much better than the 8-bit processors. Of note also was the disparity in the results obtained with different compilers on the same machine. Some of the differences could be explained due to the fact that some compilers generated interpretive code, but even among the native code compilers there was sometimes as much as a factor of 10 difference. Unfortunately, no attempt was made to explain these differences.

Titus and others [Titus81] show the code for bubble sort, string search, square root approximation and table lookup benchmarks for the 8086, Z8002, LSI-11, 9900, 68000, and NS16032. These are referred to by Motorola [Electronics, July 28, 1981, p. 72] as the Blacksburg Group benchmarks.

[Interface Age, June 1980] describes the results of running a prime number algorithm (division method) with Basic interpreters for various machines. A bipolar bit-slice processor was the fastest.

8. ADA LANGUAGE ISSUES

The Ada language has been examined for areas which could cause implementation difficulties or which require special attention. The following discussion attempts to delineate some of these areas.

8.1 Data Space Management

The Ada language requires an underlying mechanism to perform data space management. This space management is separated into two distinct functions although at some level they may conflict in their request for space.

The first function is used to provide the local storage for subprograms and blocks. The algorithms are quite simple; space is acquired upon entrance to the subprogram (block) and released upon exit or termination. Since this space acquisition/release is well-behaved, any of the usual stacking mechanisms accommodate this type of space management and are quite efficient. There is little compiler or run-time library impact associated with this space management except in connection with parallel paths (addressed below). However, a display is required for accessing recursive outer scope data and a mechanism for dealing with a stack overflow must be present. Neither imposes significant overhead upon a subprogram (block) entrance or exit.

The second form of space management is much more complex. This space is used in connection with access variables and is acquired from what is commonly known as a "heap". Although an explicit release feature exists in the language, the default semantics are that this space may be released only when there is no access variable which can reference the space. A mechanism must be provided then to free unattached space and/or compact the storage to consolidate discontiguous holes to satisfy requests for larger chunks. Significant overhead is usually associated with the management of this space. This overhead may be handled differently depending upon the constraints and characteristics of the target embedded computer system. Some alternatives and related comments are presented below:

1. Access variables may be linked together with like typed variables to permit delinking and release of unattached data at reassignment. This requires a significant data space overhead for large lists.

2. Counts of access variable copies may be kept with the space to determine unreferenced space. A limited count field minimizes the data overhead and accommodates most lists but not multiply threaded lists. A maximum value inhibits further incrementation and, of course, decrementation.

3. Garbage collection may be performed at space exhaustion to trace all

access variables, mark all referenced space and free released space and/or compact referenced space. A severe response time penalty occurs at space exhaustion that is probably unacceptable in any real time system.

4. Incremental tracking and marking may be done to permit continual free space release. Current studies limit the overhead for incremental collection to twice that of collection at exhaustion. The tradeoff between total CPU usage for the former and the delay associated with the latter must be evaluated for the application.

5. Explicit releases may be generated at exit from a scope containing the access type declaration.

6. A transaction file may be maintained which permits incremental updating of reference counts for determining attachment. A compiler may optimize out many of the transactions by flow analysis but the overhead of the file operations are still significant.

All of these algorithms involve varying degrees of data space and code execution overhead that may not be ignored. Most of these algorithms have been developed to support well defined data structures controlled totally by an underlying language mechanism. Imposing a mechanism such as this on real time computer systems with program overlays, a high percentage of space allocated to list structure, and list structures that may be moved partially or entirely to secondary store at the application discretion (such as with a store and forward message switching system) dictates that some facility must be supplied to minimize space release and automatic reclamation by garbage collection. Although the potential use of dangling pointers is undesirable, system constraints may require such treatment of this space.

A complication derives from space management with concurrent processing paths (Ada tasks). In addition to the requirement for multiple stacks (or acquisition of stack space from the heap), certain mutual exclusion areas are necessary within the space management routines. Note, however, that these may already be present for an incremental collection scheme. If stack space and heap space are acquired from the same space pool, the stack space need not enter into the garbage collection process and should be distinguished as requiring a release. This release may be prompted invisibly when exiting the closed scope which had requested it by inserting a return to a space interface routine between the caller and callee requesting the space extension.

If heap space compaction is required, all tasks using the heap must be suspended to pack the used space and adjust any access variables, including those contained in registers. This further places a burden upon optimization and the space management mechanism.

8.2 Input-Output for Ada Programs

Since input and output in Ada will be done by calling procedures defined in packages, the compiler will not need special knowledge about input or output. It will be possible to generate code as if INPUT_OUTPUT and TEXT_IO were any other kind of package. Some issues related to the implementation of these packages need to be resolved, including the method of implementation of objects. The intention of SET_NEXT, acccording to the Ada Rationale, is to allow random access for those objects to which it is applicable. CREATE, however, does not specifically provide an indication of whether the object created should have sequential, indexed sequential, or random organization. The file organizations for devices such as disks must be decided upon. Sequential access is sufficient for a number of applications. Paired SET_NEXT and READ or WRITE calls give a form of random access. If the calls are not paired, but multiple READs and WRITEs follow a SET_NEXT, the behavior is more like indexed sequential. Also, successive reads may result in discontiguous records being read, if there are undefined records in the file. Thus, a case can be made for supporting all three organizations.

The language does not specify how the user should indicate the desired file organization, or whether this should be permitted. A number of possibilities exist. Files may be organized in a manner which permits facilitates the types of accesses described above. One example of such a structure is the B-tree [Knuth73, Comer79]. Another possibility is to allow the user to specify file organization, either in the command language or in the file name parameter to CREATE.

8.3 Text I/O for Ada Programs

Input and output for the files will also be done by calling TEXT_IO. Since the text input and output procedures will be able to use the procedures in the package INPUT_OUTPUT, the issues which apply to INPUT_OUTPUT tend to apply to text input and output.

Enumeration types have a generic package because the REP attribute is different for different enumeration types. It would probably be possible to have only one copy of the code for the enumeration type text input and output routines, with a different table for the REP attribute for each instantiation. Whether such an optimization is worth the effort is a design decision.

8.4 Tasking

Since the rendezvous as a parallel processing language concept has only recently become popular (although its roots go back to the 1960's), there are a number of issues related to tasking which will need to be resolved during the design phase. Most of these issues concern the implementation of tasking on the target and only affect the output of the compiler and not compiler structure.

The implementation of the select statement has several possible strategies, especially when the guards have no side effects. The brute force method evaluates each guard each time the select is executed. Other strategies (some are given in the Ada Rationale) allow a more efficient implementation.

The select statement is likely to be one of the more interesting language features to implement. There are several reasons for this. First, there are three varieties of select, each with different syntax and semantics. Second, the order of evaluation is quite different from the order in which the components of the select appear in the source. Third, in some cases the rules for the selection of alternatives require that all open alternatives be examined. This requires a certain amount of intelligence on the part of the implementation.

Two other areas which are bound to create interesting implementation issues are the presence of tasks in recursive procedures and the means of passing parameters to accept statements.

The design for tasking facilities that is sketched in the Rationale is more or less a proof of existence and is not intended to represent the most efficient implementation. Thus, the design effort will have to concern itself with how to implement tasking efficiently. One aspect of this is how to interface with the host operating system. Should the host's tasking facilities be used by the framework or should the framework have a tasking machanism which is completely independent? A possible problem area is the necessity for operations involving the updating of the queues to be uninterruptable. For an APSE running under a host operating system, the definition of uninterruptable can be stretched to mean not interrupted by another task in the APSE system. Nonetheless, caution must be used to insure that there is no chance that queue updating operations are incomplete when a dependent attempt is made to access the queue.

Special consideration must be given to exceptions raised in the presence of multiple tasks. FAILURE is treated as a special kind of exception. It is the only exception which may be raised in one task by another. It is not propagated to the calling task by a caller which receives a FAILURE raised by a third task. Rather, a TASKING_ERROR is propagated.

An exception raised in an accept statement but not handled there is propagated in both the calling and called tasks. Also, if the called task is terminated before the rendezvous is completed, the exception TASKING_ERROR is raised in the calling task. The system must be able to handle these situations.

Since tasking is a feature which is likely not to appear in a large number of programs, pains should be taken to insure that the cost of tasking is as small as possible for those programs which do not use it. This applies both at compile time and execution time.

Because subsets of Ada are not to be considered, it is necessary to design a compiler which can handle the tasking constructs such as selects, accepts and entries. However, it would be wise to handle such constructs in an optional phase or phases so that the cost of tasking is negligible for compilations of programs in which these constructs do not appear.

8.5 Exception Handling

The exception facility is a feature of Ada which has an impact on both the compiler and the linker. Exceptions are only supposed to add to execution time in the event that they are raised. The Rationale sketches a method for achieving this goal.

Since an exception may be propagated beyond the scope in which its name is known, unique codes must be assigned to exceptions across an entire link. For this reason the compiler must communicate information about exceptions and their handlers to the linker. This, of course, implies that there must be a special Ada-oriented linker, since existing linkers do not support such features.

8.6 Generics

Generic program units are only templates and, therefore, differ from ordinary program units. Several interesting issues arise from the generic feature.

Since the rules for generics require that resolution take place at the point of generic declaration, rather than instantiation, it is necessary to retain an IL representation of the program for any generics that are visible from other compilation units or subunits.

Also, since there are expressions in the generic part which must be evaluated in the context of the declaration elaboration, but which are used by the instantiation, the compiler must provide a means by which the instantiation can use the appropriate values. This can be done by creating additional

constant or renaming declarations at the point of declaration, and referencing the appropriate objects in the IL for the instantiation.

Generics and resolution present an ordering problem for a multi-pass compiler. It is necessary for resolution to be performed on the generic's body at the point of declaration. This means that resolution must precede instantiation, unless the context of the declaration is to be reconstructed for a special form of resolution after the generic is instantiated. (This would be a funny form of "macro" expansion and would be quite inefficient). However, the instantiation of a package can create subprogram, type, and object definitions which may be referenced by program units which follow the instantiation. This implies that instantiation must precede resolution.

Note that this is not a problem in a single pass compiler, since the generic body is encountered before the instantiation, which, in turn, precedes uses of entities by instantiation. Thus, resolution is performed on the body of the generic before it is instantiated; the generic is instantiated before its declarations are referenced outside its own body and resolution takes place using the instantiated definitions. It is tempting to perform resolution and generic instantiation in separate phases, but this leads to the ordering problem described above. Resolution and instantiation must be performed in the same phase to handle the general case. (The problem does not exhibit itself if the generic body, the instantiation and the use of instantiated entities each occur in separate library units.)

At both compile time and link time there are opportunities for optimization that result from generics. At compile time it might be desirable to combine generic instantiation for different types if the generated code happened to be the same. An example of a case where this could happen would be for a generic package for stacks. Assuming a reasonable implementation of the package, any types whose representations happen to be the same size could share the same code.

Not all of the optimizations related to the generic feature can be handled at compile time. Suppose two different packages instantiate the same generic unit with the same generic parameters. At compile time there is insufficient information to tell that a duplication has taken place. The linker, however, could be provided with sufficient information so that it could eliminate redundant instantiations.

8.7 Overloading

Several factors must be considered with regard to overloading. First, the analysis of expressions can be considerably more complicated as compared to typical ALGOL-like languages. It is no longer possible to determine what

operation is intended by examining just the operator and its operands. Contextual information must be passed up and down the expression tree until the types of all of the operators and operands can be disambiguated. One possible design strategy would be to attempt to handle expressions in the traditional way, except when overloaded operators were actually present in the expression. This would help avoid having to pay the cost of overloaded operator processing in the vast majority of cases.

Second, the handling of names is somewhat different. In traditional block structured languages, a duplicate name in an inner scope hides an object with the same name in an outer scope and the occurrence of the same name for different objects in the same scope is an error (with rare exceptions). With overloading, however, a procedure or function only hides the name of an outer scope procedure or function if the types of their parameters are the same. No longer is it possible to search just for the innermost occurrence of a name. In some cases all scopes must be searched to make sure that all visible overloaded functions are considered when types are being resolved.

8.8 Linking and Loading

Ada imposes requirements not only upon the compiler but also upon the linker. This is in contrast with J73 which has been implemented to generate relocatables that are processed by existing linkers. An important issue to be determined during the design are the features to be supported by the linker. Automatic library searches for external symbols, program overlays, user-oriented listings and convenient linker commands are obvious requirements.

User convenience must be a major objective of the linker design, particularly in the manner in which the linker supports configuration management. Ada dictates an order of compilation in system development which guarantees interface version integrity but could burden program development and maintenance. Minimizing this overhead will be an objective of the linker design. Default version selection will assist the user during creation of ever-evolving builds.

A feature often lacking in system linkers which greatly facilitates large system development and maintenance is a capability to perform partial links as required by the Ada language definition. To accomplish this, the linker must produce as well as accept relocatable objects. To eliminate the requirement for the linker to produce two object formats, one for subsequent linking and a different one for loading, a single one may be defined that satisfies both functions. A loadable object could simply be a relocatable object containing no unresolved symbols (or at least none that the loader could not satisfy). To minimize loading overhead, the linker could produce load address biasing

which would allow the loader to ignore object relocation information.

Another important function that should be supported by the linker and loader is the acceptance of symbolic debugging information in object modules and the association of this information with executing programs. This information should always be available for use by a debugger but should not occupy memory space except when in the debugging mode.

With the definition of a retargetable linker comes the desirability of defining a common, machine-independent object module format. Having a standard relocatable format permits the inclusion in the design of a standard relocatable object module formatter in the compiler and elimination of a previously non-trivial task required for retargeting efforts.

8.9 Optimization

There are issues in code optimization which are more or less traditional, those which are peculiar to Ada, and those which are peculiar to, or particularly important for, microprocessors. The latter two categories are emphasized here.

Possibilities for optimization to reduce a program's resource requirements exist when an object action can be expressed in any of several different ways. The increased generality of object actions in Ada and the increased security of Ada programming introduced by strong typing offer many opportunities for optimization.

Ada also provides sources of optimization information, again in the form of strong typing. Variables may be declared with sub-types constraining the range of values they may assume. This permits the selection of instruction sequences highly tailored to the operand. Additional aids to optimization are the optimize pragma, the prohibition against aliasing, and the treatment of exceptions.

Language independent optimizations are expressible as source or intermediate language transformations and are valid for most target machines and architectures. Examples are common subexpression elimination, loop optimizations, such as unrolling and fusion, dead code and variable elimination, constant folding, inline substitution and code motion.

Machine independent optimizations are valid for most architectures, but concern the use of resources on a target machine, including such factors as minimizing the number of accesses to memory and making use of multiple or parallel computational facilities. Examples are register allocation, variable overlaying, order of expression evaluation, Boolean expression and other

peephole optimizations, structure alignment, subscript linearization and strength reduction.

Machine dependent optimization makes the best use of individual machine instructions. Examples are the use of increment instructions, shift instructions, and addressing modes. Machine dependent optimizations are normally performed during code generation or in a peephole optimizer which executes after code generation.

The selection and specification of the optimizations and the messages generated during optimization must be considered. A major benefit of optimization may come from the diagnostics produced by an optimizer. For example, if a large section of code is unreachable, the source programmer should be advised exactly which lines of his program were deleted. However, the programmer should not be inundated with messages advising him of the deletion of code generated by the compiler, for which he has no direct responsibility.

Since some optimizations have little benefit relative to their cost at compile-time, it should be possible for the user to select optimizations and perhaps to specify those parts of the program which are to be optimized. These options should be expressible in pragmas embedded in the program and in the JCL which invokes the compiler. In the event of a conflict between the JCL and the pragmas, the JCL should take precedence because of its later binding time.

It is crucial that a cost/benefit analysis be done before deciding what optimizations to include. The optimizer will be squeezed three ways. It must be fast, small and generate good code. Unfortunately, these are conflicting goals. It is better to do a good job on the common cases and to save the space and time involved in checking for, and performing some of the rarer optimizations.

A number of studies [Knuth71, Elshoff76, Tanenbaum78] (including this one) have shown that expressions tend to be relatively short. Thus, it makes more sense to optimize the common cases like I=J, I=0, and I=I+1, rather than to attempt to generate optimal code for complicated expressions. Tailoring the sorts of optimizations the compiler performs to suit the applications for which the compiler is intended should help. One of the purposes of gathering statistics on the J73 compilers was to determine what operations need to be optimized to generate good code for the compiler itself. A similar sort of thing could be done for applications programs or even a specific class of application programs.

Other areas to be considered in optimizer design include performing constant arithmetic in host- and target-independent fashion, provision for internal

optimizer tracing and dumping of data in user readable format to permit effective optimizer debugging, and creation of test cases.

The frequent occurrence of sub-type value and index-range- checking in Ada programs imply that extensions of constant folding and propagation of range information, should be attempted. On a microprocessor the optimization of subscript and range checking is even more important than it would be on a large machine. There is great potential for optimization, particularly with regard to the subscript range checking, since the same subscript is often used to access different fields within the same record. These optimizations have the fortunate property that they save both space and time.

Recognizing the scope of an IF statement, if the IF-expression is true, leads to additional common subexpressions. The optimization of sub-type and index-range checking is somewhat different from ordinary optimization of conditionals since a failure of one of these checks generates an exception, which implies that if the condition is true, the scope containing the condition is exited. Thus, it is only necessary to make such a check once if the first check back dominates the others. This is not true for ordinary conditionals.

Consider the following example:

```
rangecheck(arraybounds,I) ;
statements;
rangecheck(arraybounds,I);
statements-2; .
```

This may be optimized to:

```
rangecheck(arraybounds,I);
statements;
statements-2; ,
```

since it is known that if "statements" is executed that no range error can occur at the second rangecheck.

On the other hand;

```
IF cond(I) THEN statement-a
statements-1
IF cond(I) THEN statement-b
statements-2
```

cannot be optimized in the same way. The cond(I)'s may be found common but the second IF may not be deleted, since statements-1 will be executed regardless of whether the condition is true or false. (This assumes, of

course, that statement-a is not a goto).

Unfortunately, optimization is most needed, but also most difficult for small machines. Optimization is needed because programs must be compact in order to fit in the machine. In addition, since the hardware is not as fast, inefficiencies in the generated code will be more apparent in reduced performance.

A prime example of an optimization which is more important on a microprocessor than on a large machine is strength reduction. On a machine with a hardware multiply, the effect of changing a multiplication to an addition is to save a few microseconds per execution of a loop at the cost of several words for the initialization. If the machine has no multiply, the savings can be significant in terms of speed since a single machine instruction is being substituted for a subroutine call. There is even the possibility, if all of the multiplication in a program can be removed, that the library routine which performs multiplication can be omitted, yielding a savings in terms of space.

Optimization is more difficult on a small machine due to time and space constraints. Global optimization traditionally requires large amounts of both time and space if a good job is to be done. The more information an optimizer has, the better job of optimization it can do. This information includes such things as set/use data and control flow information. Room is required to maintain the data structure which contains this information. If it is kept in main memory, a larger region is required. If it is kept on disk, a time penalty is incurred. It is, of course, possible to make optimization optional, but if the optimizer is excruciatingly slow, it will not be used for normal compilations. If this is the case, it is unlikely that the optimizer would become stable within a reasonable amount of time, causing users to shy away to avoid potential optimizer bugs.

While numerous papers have been written regarding the optimization of generated code, little attention has been given to the problem of developing effective algorithms which do not require much space.

Also, optimization has traditionally attempted to maximize speed of execution, sometimes at the expense of space. Some optimizations, such as the compile time evaluation of constant expressions, provide both time and space improvements. Optimizations such as loop unrolling sacrifice space for the sake of speed.

If a microprocessor is the target machine, regardless of the host, the emphasis for most applications should be on reducing the space requirements of the object program. There is a pragma in Ada to indicate whether optimization is to emphasize space or time improvement. However, the compiler must have space optimizations built in if the use of the pragma is to do any good.

One of the common characteristics of the newer microprocessors is the availability of various methods of addressing data. Substantial savings are possible if a short form of addressing can be used for the most commonly referenced data. Often a short address can allow one word rather than two to be used for a memory reference.

In order to take full advantage of such features, it may be necessary to allocate data so that the more frequently referenced data may be accessed using a short form of address. This requires that a count of references to each data item be maintained and that operands not be bound to storage locations until the count has been tallied.

A number of machines have a feature which is sometimes called "register indirect" addressing. This is a degenerate form of indexed addressing where the register contains the address of the location referenced. The effect is the same as indexing with an offset of 0, but since the 0 is assumed, no space is required for it.

This feature can be used to greater advantage if names are found common. The method for doing this would be similar in some ways, but not identical to, finding values common. The principal difference lies in the spoiling rules.

Another machine feature which is found more often on microprocessors than on larger machines and which affects optimization is memory-mapped I/O. This machine feature, however, does not present opportunities for optimization. Rather, it requires that the optimizer use extra care if it performs certain optimizations. Common subexpression elimination and dead store suppression must not be performed if the operands which are being optimized are in the address space reserved for I/O. Performing those optimizations on operands in that address space would cause the I/O characteristics of the program to be altered.

8.10 Recompilation

One of the significant differences between Ada and other languages is the explicit requirement for compilation order checking by the compiler [Ada 10.3]. While there is an implicit requirement that the caller and callee agree on a calling sequence in other languages (e.g., FORTRAN), there is no requirement that the compiler check this. Indeed, the only source of such a requirement is that the linked program will not work (in general). The burden of making certain that the caller and callee are in agreement falls on the user.

In J73 a user may achieve a certain degree of checking by including the compool containing a procedure s specifications in the compilation which

includes that procedure s definition. If there is a discrepancy, the compiler will issue a diagnostic. There is, however, no requirement that the linker check for consistency of compool usage between modules. Moreover, it is possible to reference an external procedure with a DEF, bypassing the compool mechanism entirely.

It is the intent of Ada to relieve the programmer of the responsibility for consistency checking by moving the responsibility to the tools. It is necessary for the tools to inform the user of inconsistencies in compilation order both at compile time and at link time.

We feel that automatic recompilation is too inflexible and potentially wasteful of system resources. A compilation order problem discovered by the linker or the compiler may, in fact, signify something more than a mere failure to recompile units which depend on other newly compiled units - it may mean that a change has been only partially implemented.

In order to provide greater flexibility, but still avoid placing too much of a burden on the programmer, the detection of compilation order errors is loosely coupled with recompilation. The user is given a list of modules which need to be recompiled in the correct order for recompilation. This list may be used to drive the recompilations; however, no recompilations are performed automatically by the system. (A command procedure for doing this could be written, though.)

9. MINIMUM CAPABILITIES OF A MICROCOMPUTER DEVELOPMENT SYSTEM.

In order to determine whether a microprocessor is suitable for hosting a software development system, it is necessary to define the requirements for such a system and to evaluate microcomputer hardware to see if those requirements are attainable.

The capabilities offered by a micro-based development system will very likely have limitations when compared to the large time sharing development systems currently in use. If the limitations imposed by a micro based system were too severe the system would not be useable; for this reason it is important to establish some minimally acceptable operational characteristics for a commercially useful micro based development system.

It should be noted that the functional capabilities of some of the current 16-bit micros meet or even exceed the capabilities (per user) of some of the present large time sharing systems.

The micros do not have the raw computing speed of the larger systems nor, more importantly, do they have the operational systems software. Also, their I/O capabilities are not as great. This is likely to be the bottleneck which limits user capacity.

The major areas of concern for a program development system are the following:

System Disk Storage Requirements

System Memory Requirements

Compilation Speed - Lines/Minute

Source Program Size Limitations

Available User Disk Storage

Printer Speed

Other Peripheral Support

Multi-User Capabilities

System Facilities

9.1 System Disk Storage Requirements

The total amount of disk storage required to hold the executable modules of the system itself is very important in the overall configuration of a micro based development system.

Under "system modules" the following general functions are included (with expected storage requirements in bytes):

Function	Disk Storage
Resident System	50K
Compiler	1000-1500K
Editor(s)	50K
Linker	20K
Debugger	30K
File Utilities	50K

With these estimates, a minimum system will take approximately 1.5M-2M bytes of disk storage. This is about the capacity of 3 to 4 single-sided dual density 8" floppies, and less than the capacity of a single cartridge or Winchester drive.

The compiler estimate above is for an optimizing compiler for the full language. In comparison, the TeleSoft Ada compiler requires only about 300K bytes.

Note that these figures do not include disk storage for the application. While the amount of storage required will depend on the size of the application being developed, it would be wise to allow for another megabyte of on-line storage.

This would bring the disk capacity up to 2.5-3M. We currently feel that the 8" floppy disk will be the standard disk device for the transfer of data and programs between micro systems. However, while it would be possible to have a (bare) minimum system with only floppies, the increased performance and capacity of hard disks would seem to dictate the use of a hard disk as the primary disk.

9.2 User Disk Capacity and Printer Speed

Both user disk capacity and printer speed are more an option of the user than they are a system specification. Some important objectives are that the system not arbitrarily limit the number and type of disks which can be supported, and that the system support reasonably high speed printers.

Thus, while disk and printer characteristics are of critical importance to the user in configuring a micro system, both from the point of view of cost and of system capabilities, this area should have relatively minor impact on the design and development of major portions of the system itself.

The various types of disks which should be supported are:

Type	Size	Typical Capacity
Floppy	5 1/4" and 8"	Up to 3MB
Winchester (fixed)	5", 8", and 14"	Up to 160MB
Cartridge (removeable)	Single/Dual Platter	Up to 32MB

9.3 Compilation Speed

Compilation speed is an important factor in program development.

A development system would be impractical if compilation speeds were too slow; even where compilation speeds are barely "acceptable", a slow compiler makes program development more difficult.

Current J73/I compilers running on the IBM 370, Univac 1108 and DEC-10 have compilation speeds ranging up to 6-7 thousand statements per CPU minute. From the user point of view the apparent compilation speed in elapsed real time varies from several hundred to several thousand statements per minute, depending on system load. (See Appendix C.)

A minimally acceptable compilation speed on a micro based system is two hundred statements per minute of elapsed time. This figure was arrived at by taking 5 minutes as a acceptable delay for compiling a 1000 line module.

This is of course a somewhat subjective figure. However, current PASCAL compilers on 8-bit micros meet this compilation speed and we feel confident that even though Ada is a much bigger language, that this compilation speed can be met or even exceeded depending on the micro configuration.

9.4 Other Peripheral Support

A variety of different peripherals can be attached to current micro-based systems. A micro-based program development system should support such

peripherals or allow for the extension of the basic system to support such peripherals.

Possible peripheral devices would include the following:

Terminals
   CRT's or hard copy (300 and 1200 baud modems)
Printers
   Line oriented devices
Floppy disks
Winchester disks
Cartridge disks
Magnetic tapes
   Standard 3/4" tapes
Paper tapes
Cassette tapes
Other Computers (via dial-up telephone link)

A minimal micro development system might support the following:

1 Terminal CRT Screen
1 Printer
5 or 6 8" floppy disks

A more practical minimum system would include a hard disk. It is expected, though, that data transfer between Micro systems will normally be done via floppy disks.

9.5 Source Program Size Limitations.

A micro based compiler will probably have some source program limitations which are more restrictive than those of a compiler running on a large time sharing system.

Some estimated minimal limits, derived from a production-quality J73 compiler, are given below.

LIMITS, CAPACITIES, RESTRICTIONS

* The maximum number of levels of nesting of PROCEDUREs, FUNCTIONs, IFs, LOOPs, CASEs, and BLOCKs is approximately 50.

* The maximum number of symbols within a language construct terminated by a semicolon is 30.

* The maximum number of procedures per program unit is 100.

* The maximum number of levels of nesting of INCLUDE pragmas is 10. There is no limit on the total number of INCLUDEd files.

* The maximum number of unique names allowed in a program unit is approximately 2000.

* The maximum number of diagnostic messages, of severity warning or greater, allowed for a compilation is 200.

* The maximum number of dimensions allowed for an array is 7.

* The maximum number of characters in an identifier is 31. If an existing linker is used (this is possible only for J73), a further restriction may be imposed on external names.

* The maximum number of parameters in a procedure call is 20.

* The maximum number of data declarations is 1500.

It is an important compiler design consideration that any size restriction which canot be extended with extra memory be as liberal a restriction as possible. For example, it would be wrong to limit the number of unique names to 1023 independent of the amount of memory available.

9.6 System Facilities

The functions supplied by the operating system and its basic support programs are of critical importance to the user in any program development system.

The following minimum set of functions would be needed.

Editor
- Line Oriented (Optional)
- Screen Oriented

Compiler Ada/J73

Linker/Loader

Run Time Execution/Debugger Package

File Utilities (Data Base Manager)

Document Formatter

Operating System Functions

Command Language Interpretation

Basic I/O

Terminal Interface

Tasking

Exception Handling

A more elaborate system could allow such features as spooling, background jobs or multiple users.

To a large extent, the characteristics of the host machine and operating system will dictate the design of such tools as the compiler. The size of the memory available to an individual user is, perhaps, the most significant single influence on compiler structure. To a certain extent additional partitioning, beyond that which would be present on a large system, is desirable. However, too small a memory may dictate that phases be split artificially, rather than logically. Although it would be possible to have a microprocessor system with hundreds of thousands of bytes per user, such a machine would not be practical at the present time. If the cost of hardware should decrease sufficiently with respect to software costs, such a system would become practical.

10. SYSTEM OVERVIEW

We have already discussed a number of requirements for a development system for small and medium-scale projects, as well as the advantages and disadvantages of hosting such a system on a microcomputer. In this chapter we examine how the advantages of microcomputers can be maximized and their disadvantages minimized; we compare the requirements for microcomputer development systems to some current designs; and we give an overview of the design for a Microprocessor Development System (MDS).

10.1 Environmental Differences

A number of advantages and disadvantages of microprocessor development systems have already been discussed. We now address the effect of these differences on the development environment. Batch environments will be ignored; the comparison will be that of an interactive environment on a large computer versus an interactive environment on a microprocessor.

Two of the more important differences are with respect to communications between the user and the machine. On a large system there may or may not be access to the machine by means of high-speed video display terminals (VDTs). If dial-up lines are used, communication speeds will be lower. This prevents the use of display-oriented editors and provides a strong disincentive to "browsing" through source files, both of which are possible on high-speed terminals.

A large computer system is almost certain to have a high speed printer, but a microcomputer is less likely to have one. The effect of this is to encourage obtaining listings on larger systems, but to encourage perusal of source files on a microcomputer system. (On a large system which has both high-speed VDTs and a high-speed printer, either development style is applicable. The same would be true on a microcomputer system which had a high-speed connection to a host which had a high-speed printer.)

The tools available on a development system which lacks a high-speed printer should make it possible for development work to proceed in the absence of up-to-date hardcopy listing. There are a number of facilities which the system can provide to make this more convenient. Perhaps the most important feature in this regard is a screen-oriented editor. This tool allows the user to examine code in context to insure that the source is as it is supposed to be, and to find programming errors. An extremely useful facility that the editor can provide is a means for relating compiler diagnostics to the source code. This could be done by having the editor display the diagnostics which were associated with the lines which appear on the VDT. In addition, the

compiler can produce information which allows the editor to find the next line containing errors. This is a generalization of a UCSD Pascal system feature which allows a user to invoke the editor from the compiler. The compiler provides the editor with sufficient information so that the cursor is positioned to the point at which the errors were detected.

Since the user is not as likely to have a current cross reference listing available as on a system with a fast printer, it makes sense to provide a cross reference tool which allows the user to ask questions about the uses of a given object. One option would be to produce an entire cross reference listing. Other options could list the references to a given variable.

Because it is to be expected that obtaining listings will be the exception, rather than the rule, it makes sense to provide separate listing tools (prettyprinter, cross reference generator, etc.) rather than including such facilities in the compiler. Although the cost for doing a compilation and obtaining a listing will be greater in terms of machine resources than if the listing facilities were integrated into the compiler, the cost of a compilation without listings will be less, since the compiler will be smaller. This is particularly important in the case of Ada, since the Ada compiler is, by far, the largest tool in the MDS system. (This is in contrast to Pascal or Basic compilers which fit comfortably on systems of 64K bytes or smaller.)

There are two approaches which could be taken with regard to the listing tools. One is to construct the listings from the source; the other is to work with an intermediate level representation of the program. The disadvantage of the former is that the source must be retokenized and reparsed. The disadvantage of the latter is that an intermediate representation of the program must be maintained. This would require extra disk storage for a given project.

10.2 Distributed Computing

One of the more important differences between microcomputers and mainframes is that it becomes economically feasible to have a single user per CPU. Two alternatives to the traditional timesharing approach are having multiple CPUs share a common memory and having a network of CPUs with separate memories. (Note that both of these schemes have been used for large machines, but not on a single-user per machine basis.) The advantages of having only one user per CPU is that the operating system is simplified, and that CPU response is independent of the number of users of the network. Ada tasking is not well suited to multiprogramming since Ada task specifications must be compiled before their caller. In the case of an operating system, the "tasks" are programs whose specifications may not even exist when the operating system is

compiled. Having only one user on a CPU means that the only scheduling problems are those associated with Ada tasking. There must be, however, a facility for communicating with a global data base. Such a system would resemble that proposed by Intermetrics for the AIE, the differences being that rather than having a virtual machine for each user and one for the database, there would be an actual machine, instead. In addition, it would be possible to give each user a local backing store in such a distributed setup.

10.3 Relation to Ada Integrated Environment (AIE)

At the time that the Microprocessor Development System was being designed, the three designs for the Ada Integrated Environment (AIE) [CSC81, Intermetrics81, TI81] had been submitted, but the winner had not been chosen. We anticipate that the winning implementation, although based primarily on the winning design, will, nonetheless, be influenced by some of the features of the other two designs.

The Ada Language System (ALS) [Wolfe81] is an Army effort parallel to the Air Force's AIE. Although the implementation contract has already been awarded, the design documents for the ALS did not become available soon enough for the ALS design to have much influence on the design presented here.

Assuming that there is an implementation of an Ada environment for a microprocessor based system, it would be to the Government s advantage to insure that there is as much commonality as is practical between the three systems. It would be desirable for the facilities provided on the microprocessor system to be a proper subset of those provided by the AIE and ALS. This would allow programmers to move back and forth more easily between the two systems.

Because the final AIE design has not been formulated, we have not attempted to make the microprocessor system a subset of any of the AIE designs. Rather, we have borrowed from all three. Admittedly, the influence of the CSC/SEA AIE [CSC81] is the strongest, since the authors of this report also participated in that design. The Intermetrics/MCA design [Intermetrics81] and the Jovial Users Group (JUG) AIE evaluations [JUG81] were also influential.

Both Stoneman and the AIE statement of work were used as guidelines, rather than as absolute requirements.

The operating system for the Microprocessor Development System must provide all of the basic services which an operating system provides. These include the ability to load and execute programs, job scheduling, and primitives for performing input/output. Other functions which are necessary for program development are supplied by such tools as the editor, compiler and linker.

Because the MDS may be physically isolated, security can be derived to a greater extent from the physical surroundings than would be possible for a large timesharing system. Since malevolent users could be prevented from accessing the system by excluding them from the area in which the system and terminals are kept, it is feasible for the MDS to largely ignore the problems of pathological users. Of course, it is still necessary for the system to provide some safeguards against user errors, by providing file backup facilities, read-only flags and the like. The approach taken in the design of the MDS was to protect users against inadvertent damage to data, but to ignore the problems presented by hostile users.

There are several other important differences between the AIEs and the MDS environment. One is that the AIEs are systems on top of systems. There is, in each case, an underlying operating system, although the way in which the AIE interacts with the host system differs from design to design. For example, in both the TI and Intermetrics/MCA designs, each user is given a separate virtual machine but in the CSC/SEA design, one virtual machine serves a number of users.

Another closely related difference between the AIEs and the MDS is that the AIE designs assume that a given host will serve non-AIE users as well as AIE users. We do not make this assumption in the MDS design. At least there is no provision for MDS and non-MDS users to use the same machine concurrently.

Thus, the MDS is the actual operating system as well as a virtual system. The user interface is the same, regardless of the host, so there is a uniform virtual interface, but there is no operating system below the MDS itself.

10.4 MDS Components

The MDS consists of the following major components: the operating system, the database and associated utilities and the tools. The set of tools is, of course, open-ended, so each instantiation of the MDS can be expected to have its own specialized tools. However, there is a subset of tools which are vital to any MDS. These include the command language interpreter, the compiler, the linker, the debugger, and the text editor. Each of these is discussed in some detail in subsequent chapters. Other tools of somewhat lesser importance, such as the cross-reference generator and the lister are discussed in less detail.

For each system component, a design rationale is provided along with the specification. The System itself and the major tools are described in a pseudo-B5 format; the Quality Assurance section was omitted, and the specifications are somewhat less formal than a genuine B5 would be. These specifications are contained in seven accompanying volumes: MDS Operating

System, Database System, Command Language Interpreter, Ada Compiler, Linker, Debugger and Text Editor. The compiler has received the most attention, since, according to the statement of work [SOW], it was supposed to be emphasized. The minor tools are described less formally in Chapter 18 of this report.

10.5 System Structure

Whether or not the system should be of a distributed type is one of the fundamental issues with regard to the system design. The fact that microcomputers are so much cheaper than larger-scale computers makes realistic, configurations which would have been unthinkable with more primitive technologies. In particular, it is realistic to consider giving each one his own CPU. An isolated CPU with a single-user system is unrealistic for serious development work, but it is possible to arrange single-user work stations in a network so that users may communicate with each other.

There are advantages and disadvantages to both the network and more traditional multi-user configurations.

The single-user workstations communicate with each other via a network. It is possible to have configurations in which each user has local secondary storage or where all secondary storage is on a common database machine. The advantages of such a system are as follows:

1. The operating system is simpler; the development cost would be lower.
2. There is less system overhead (time and space).
3. There is likely to be better CPU response.
4. Each workstation can serve as a backup for the others.

On a single-user machine the operating system on each work station can be simplified, since it is not necessary to address issues such as context switching, protection of one user from another and database synchronization issues except with respect to Ada tasking. The database machine would have to address issues such as preventing two users from accessing a file in a conflicting manner. Most of the more recent microprocessors distinguish between system and user modes, so it is relatively easy to protect the system from the user. There is not always a simple means, however, for protecting one user from another.

Since the operating system has less to do, there is less system overhead in terms of both time and space.

In practice there is likely to be better response to user commands with single-user workstations because each user has his own CPU. This will certainly be the case if there are two or more users on a multi-user system. While it would be possible, in theory, to provide each user with his own multi-user system and to arrange them in a network, it is unlikely that this would be done in practice, since this would incur the overhead from both the single and multi-user strategies.

Another advantage of single-user workstations is that if a problem develops in one, the other workstation can still function. In a multi-user system, all users of the system would be prevented from using the system if it went down.

There are areas, however, in which the multi-user strategy is preferable. These are as follows:

1. The ability to run background jobs
2. Lower hardware cost
3. More flexible use of resources

The multi-user system already has the capability to run multiple processes, so it is relatively easy to add the ability to execute jobs in the background. This is useful in that a user can perform highly interactive work in the foreground, while time-consuming jobs are executing in the background. Printing, document formatting, compiling, linking and other such tasks can be performed without preventing the use of the terminal. Spooling could be done easily by the database computer in the workstation strategy, but adding the capability to run general background jobs at the workstation would eliminate the advantage of having a smaller system for a single user.

There is also a lower hardware cost associated with a multi-user system. In order to run a large job on a work station, the workstation must have enough memory. On a multi-user system, the total amount of memory may be smaller if most programs are small, because only as much memory as is actually being used at a given moment need be allocated to a user. The network requires hardware for communication and additional CPUs as well.

The additional hardware cost is becoming smaller, however, and in a few years may be so small relative to software costs as to be negligible.

The fact that resources may be used more flexibly on a multi-user system is closely related to the lower hardware cost. Memory, in particular can be allocated as needed. Also, it is possible to add users more easily, since all that would be required is a port and a terminal. There would, however, be a response-time penalty.

11. OPERATING SYSTEM

The operating system must provide a number of facilities for its users. In addition to protecting users from one another and allocating resources, the system, together with the runtime library routines, provides a virtual machine interface at a higher level than that provided by the hardware. It is possible to keep this virtual interface constant for various host machines, enhancing the portability of programs which run under the MDS.

11.1 System Functions

There are a number of classes into which system functions fall:

1. Login/logout
2. Input/output
3. Get/release space, resources
4. Process handling
5. Ada tasking support
6. Date, time accounting
7. Program load
8. Debugger interface

Login and logout serve primarily to associate a user with a process. As part of the login procedure an initialization program may be run. Logout serves to disassociate a user with a process. Strictly speaking it is unnecessary, but having a specific logout helps configuration management in that the next user of a given terminal is required to provide identification.

Input and output support provides low-level support for the packages INPUT_OUTPUT and TEXT_IO. The system input and output routines are not generic; operations are on files of packed arrays of bits.

I/O support procedures exist for each procedure defined in the package INPUT_OUTPUT as well as the following procedures:

1. Pipe - establishes a pipeline
2. Standard_input/output - returns the name of the standard input/output object, which may be used as an argument to create.
3. Current_input/output - returns the name of the current input/output object.
4. Set_input/output - sets the name of the current input/output object to its argument.

Terminal I/O support procedures exist so that tools can perform I/O to

terminals in a terminal-model-independent fashion. Note that I/O to a terminal which is being used as a sequential file is performed exactly as I/O to a disk file, from the caller's point of view. However, certain functions (e.g. clear screen) are meaningful only if the device is a VDT. It is for these sorts of functions that the terminal I/O support provides a uniform interface.

The get and release space routines allow user programs to expand and shrink within the limits of memory.

Process handling facilities allow background jobs to be created, processes to be terminated, and the interrogation of the process state.

Ada tasking support provides the primitives necessary to implement tasking as described in [Ada Reference].

Date and time allow the return of the current date and clock time. In addition, there is a facility for setting the date and time.

Program load provides a means of loading and executing a user program or of loading an overlay.

The debugger interface facility allows the debugger to operate in a separate address space, but to bypass the normal protection mechanism in order to examine or modify the program being debugged.

11.2 Terminal Input Editing

The system provides a standard, rudimentary editing capability which is available to any program which does line-at-a-time terminal input. To a certain extent, this processing is transparent to the calling program. The caller only sees the final edited line; there is no difference between an edited line and one which was originally entered correctly. The process is not fully transparent, because it may not be used by programs which require input a character at a time (i.e., programs which respond to characters before a carriage return is entered).

Editing features which are available are:

1. Cursor movement (non-destructive)
2. Character delete
3. Line delete
4. Character replace
5. Character insert
6. Line echo

7. Line restore

The first five features described here allow typographical errors to be corrected. The cursor movement feature was included because errors are often not detected until after additional characters have been typed. Allowing non-destructive cursor movement means that only corrections need to be made — it is not necessary to retype the rest of the line. Characters are deleted with cancel (control-X). Characters are replaced by entering control-R, the replacement characters and the system string terminator (default "/"). The terminator character may be entered by preceding it with the system escape character (default "\"). Thus "\/" replaces a single character by a slash. The escape character may be entered by typing two consecutive escape characters ("\\"). Characters are inserted simply by typing them. They are entered at the position in the line to which the cursor points.

The line echo facility prints the line as edited. This allows the user to see the result of the editing before submitting the line to the command processor or some other consumer. Echoing is requested with control-E.

The line restore facility allows the previously entered line to be edited and resubmitted. This can be of use if the command processor detects an error, particularly if the command line was lengthy. If a control-E is entered before the first character of a new line is entered, the previously entered line is echoed and is made available for editing.

Note that the characters described here are the defaults; the actual characters used may be changed by the user.

12. DATABASE

The MDS Database is the repository for programs, data, documentation and other information. Distinguishable entries in the database are known as objects.

The MDS database provides additional features above and beyond a "bare-bones" file system, but is conservative enough so as to be implementable. The primary difference between the MDS database and an ordinary file system is the relational aspect of the database. Objects have attributes associated with them. These attributes allow relations between objects to be maintained. In addition, attributes may be used to record information about individual objects. Attributes are discussed further in section 12.2.

Microprocessor systems typically have separate directories for different devices and volumes, without a higher-level directory which points at the devices or volumes. Thus, it is usually necessary to specify a device (such as a particular disk drive) or a volume (such as a floppy disk name). This becomes inconvenient, so it was decided to include a master directory in the system, along with a search capability. This is not strictly necessary but it makes the system more useable.

Another feature that is included is automatic file allocation and linking. Allocating files contiguously would simplify the file handler, but would cause problems for users once the disk storage became fragmented. There are microprocessor systems which allocate files contiguously, and those which use linked allocation. At least one large-host system (OS/360) requires that files be pre-allocated and that disk storage be compressed when all of the space has been allocated, even if portions have been freed. This may be acceptable in a production environment, but in a development environment, where files are constantly being created and deleted, this creates a hardship for the users of the system.

Another feature of the database is keyed records. Line keys afford a fixed reference point for source text throughout the development cycle. This is particularly useful in pinpointing the source of error when the compiler is run with listings turned off. Even if a listing is obtained, having fixed keys allows the user to advance immediately, within the editor, to the point in the file at which corrections need to be made, without having to scan the text or to try to find a unique character string.

The keys can also be used to make it easier to distribute source changes for programs which are used on multiple systems. As long as the source file is not renumbered (and this should only be necessary for large-scale changes), the user's and maintainer's line numbers will be the same.

Line keys may be ignored by programs which choose to do so. Keyed files may be read as if they were unkeyed text files.

Most of the other database requirements are imposed by the Ada definition. In particular, the database must be able to meet the requirements of the packages INPUT_OUTPUT and TEXT_IO. Additionally, these packages must be augmented to allow keyed records to be processed.

No serious development system would be complete without a backup system. While Winchester disks are reportedly extremely reliable, there is still a chance of a malfunction. Since Winchester disks are usually not removable, and since their capacity far exceeds that of floppy disks, additional hardware is required in order to allow convenient file system backup. In addition, there must be software to enable files to be saved and restored. Cartridge disks do not require additional hardware for backup, since the cartridges are removable.

Files are more likely to be in jeopardy due to user error, rather than hardware failures. The file system can provide some help in this area, however. One particular method is to allow files to be marked as read-only or not-deletable. This can prevent files from being deleted due to simple typographical errors. Such a facility is valuable, is also easy to implement, and is, in fact, in existence on some present-day microprocessor systems.

It should be noted that several companies offer relational data base packages for microcomputers, although it is not clear how comprehensive these packages are.

The database provides configuration management aids in the form of attributes and tools. A discussion of configuration management requirements can be found in [Mooney81].

12.1 Utilities

There are a number of high-level utility programs which provide services for the user. These utilities may be executed at the command language level or by means of procedure calls. Utilities provided are:

1. Object copy
2. Object renaming
3. Object compare
4. List partition members
5. Create partitions
6. Add/delete partition members.

12.2 Attributes

One of the most important features of the database is attributes, which provide a number of facilities not found in conventional file systems. These include:

1. The ability to have tools handle different data base objects in different ways depending on attribute values.
2. The ability to record relations between objects in the database in a convenient, uniform way.
3. The ability to query the database in a general way.

Attributes are also used to maintain the information, such as object sizes and creation dates and times, which is normally found in directories.

There are a number of system defined attributes that are maintained by the various tools described in this report. These include:

INCLUDES - objects included by the attribute owner via the include pragma.

DEPENDS_ON - objects which the attribute owner depends on (in the sense of compilation order rules).

COMPILED_FROM - (primary) source object which was compiled to create the attribute owner.

COMPONENT_OF - objects which were linked together to form the attribute owner.

SUBUNITS - subunits for which stubs are present in the object

SUBUNIT_OF - parent unit to this subunit

VERSION - version of the object (see 12.3)

REVISION - revision of the object (see 12.3)

DATE_CREATED - date when the object was created

TIME_CREATED - time when the object was created

CREATOR - tool and user which created the object

OWNER - owner of the object (presumably the user responsible for the object)

DERIVED_FROM - objects of which the attribute owner is a descendant

Facilities exist to perform the following operations on attributes:

1. Create attributes
2. Read attributes
3. Set attribute values
4. Protect attributes
5. Delete attributes

The attribute system is open-ended. Users may create their own attributes for human use or for use by tools. It is, of course, necessary to be able to set and read attribute values. The protection of attributes is primarily to maintain the integrity of attribute values for those attributes which are manipulated by tools, for example, those which deal with compilation order. Allowing users or tools other than the compiler to modify those attributes could lead to compilation order violations which, in turn, could introduce bugs in the executable program.

12.3 Versions and Revisions

The database provides for both versions and revisions. Versions of a configuration exist for different purposes (e.g. for different target machines) while revisions represent modifications to specific versions -- a revision is temporally later than its predecessor. Revisions and versions are not strictly necessary in the sense that they could be simulated by file renaming. However, they make it easier for both the user and the operating system to keep track of the development history of an object.

The system allows both official and unofficial revisions. Official revisions are members of an established configuration, unofficial revisions are not. Unofficial revisions are assumed to be relatively short-lived, for example, the revisions which are made during the checkout of an upgrade of some program. Once a new (official) revision is created, the temporary revisions which were created since the last official revision may have outlived their usefulness.

Fewer controls are placed on unofficial revisions. They may be updated or deleted without restriction.

Both revisions and versions are attributes of database objects or library members. Since the revisions and versions will often be used to distinguish between various objects or members of the same name, a shorthand syntax is introduced into the command language to allow them to be specified more conveniently. The syntax for a version is:

: version-designator

and for a revision

official-revision [.unofficial-revision]

Note that a revision which is an official revision will be missing the "." and the unofficial number.

12.4 Program Library

The Ada program library serves as a single centralized mechanism for maintaining all of the program unit specifications and program bodies required in the development of a set of programs under that particular program library. It is used both as input to the compiler and linker and acts as the repository for recording the results of the compilation and linking of a set of related compilation units.

The program library provides a mechanism to be used in the compilation and linking of a set of related programs to insure that the consistency of the programs is maintained and that the order of compilation rules of Ada are enforced in a controlled fashion. Ada defines the program library in terms of the compiler and compilation order rules; however, including the linker function is a natural extension to the Ada program library and does not compromise the concept in any way.

A program library need not necessarily represent a "program" in the usual sense; it could, for example, contain a collection of one or more sub programs or packages used in the development and testing of other programs.

The program library is the means by which the user controls which package and sub-program specifications and bodies are available for use when compiling or linking programs.

A program library will be represented in the data base as a compound object of category 'LIB'. It contains (at least in a logical sense) program specifications and bodies.

12.4.1 Background

The output of the compilation of a compilation unit (sub-program or package) includes a program unit specification (Ada object of class 'SPEC') and possibly a program body (Ada object of class 'BODY').

The program unit specification (often referred to as the unit spec) is an internal encoding of the specifications of those Ada entities defined by the program unit.

The program body is an internal encoding of the relocatable library representation of the compiled program unit; this includes the program templates resulting from generic declarations and from IN_LINE sub program declarations.

The compilation of a compilation unit (sub-program or package) requires as input the program unit specification and possibly the program bodies of those program units referenced by the program being compiled and produces as output a program unit specification and possibly a program body of the compilation unit being compiled as described above.

It is a major concern of all program development systems that all of the program units in a system be consistent. For example, a sub-program body must match its specification and all programs which reference that subprogram must use that single correct specification.

In most present program development systems, manual procedures are used to "insure" program consistency.

In Ada, the concept of a program library has been defined to be used along with the compiler, the linker and the library maintenance tool to rigorously insure and maintain the consistency of the set of programs being developed within a program library.

The concept of a program library in Ada is a simple one - it is the set of all library units (program unit specifications and program bodies) needed for the development of the programs within the library.

A "consistent" program library is one in which all of the program units in the library have been compiled in an order defined by the order of compilation rules in Ada. The program library tool does not guarantee that the program library is consistent but rather allows the consistency of the program library to be verified.

12.4.2 Library Contents

A program library contains the following elements:

Program Unit Specifications

Unit specs are produced and initially recorded in a program library by the Ada compiler. A unit spec may have any number of associated program bodies in a library.

Program Bodies

Program bodies include both relocatable binaries and generic templates. Each program body is associated with a single program unit specification in the library.

Linked Program Units

The linker produces a program library unit which contains a program unit specification and an associated program body in the same format as that produced by the compiler. The output of the linker can be recorded in a program library just as the output of the compiler.

A program unit spec or program body may be contained in several program libraries. It is logically contained in each library, and users may manipulate the spec or body, independently of whether it exists in other libraries. However, only one physical copy exists. The system must insure, for example, that deleting a spec or body from a library only removes it from that library, and does not delete the physical entity, if it exists in other libraries.

12.4.3 Program Library Maintenance

Maintenance of the program library is carried out in one of three main ways: through the library mantenance tool, by the compiler and by the linker.

Library Maintenance Tool

The library maintenance tool allows the user to manipulate program libraries through explicit commands. Operations available through the library maintenance tool will include the following:

- Program libraries may be created.
- Program libraries may be deleted.

- Program libraries may be copied.
- New versions of program libraries may be created - this is essentially a copy.
- Program unit specifications and their associated program bodies, if any, may be copied from one program library to another.
- Program unit specifications and/or their associated program bodies may be deleted from a program library.

In normal operation, once a program library is created and initialized with the externally defined program specifications and bodies needed for the system or set of programs being developed in the library, most program library maintenance will be performed automatically by the compiler and linker.

Compiler - Program Library Maintenance

The normal input to the compiler consists of the source of a compilation unit and a program library.

All program unit specifications and program bodies required by the program being compiled must be "contained in" the specified program library.

The output of the compiler includes a program unit specification and/or a program body (objects of class 'SPEC' and 'BODY'); these outputs are recorded in the specified program library and all the necessary library dependency attributes are updated.

Linker - Program Library Maintenance

The normal input to the linker consists of link directives and a program library.

All program unit specifications and program bodies required to validate and create the linked structure must be "contained in" the specified program library.

The outputs of the linker include a program unit specificaton and a program body of the linked structure (objects of class 'SPEC' and 'BODY'); these outputs are recorded in the program library and all necessary library dependency attributes are updated.

12.4.4 Program Library Dependency Attributes

A record of the interdependencies of all program unit specifications and program bodies within a program library will be maintained by the compiler by means of the dependency attributes.

With the dependency attributes the consistency of the programs in a program library can be controlled and maintained.

Program Unit Specification Dependencies

Each program unit specification in a program library will contain a list of all the unit specifications which it depends on and a list of all the unit specifications and program bodies which depend on it.

Program Body Dependencies

Each program body in a program library will contain a list of all the unit specifications and program bodies it depends on and a list of all the unit specifications and program bodies which depend on it.

12.4.5 Multiple Versons and Revisions of Library Units

The Ada manual does not consider having multiple versions or revisions of the same compilation unit in a program library.

Since multiple versions and revisions of program units are a practical requirement it must be decided whether to extend the program library to allow multiple versions and/or revisions of the same library unit in a program library.

Disallowing multiple versions and revisions in a program library simplifies the library somewhat, since no rules need to be defined for choosing between duplicate program unit specs or bodies at compile or link time.

If multiple versions of program units are not allowed in a library, a user would be required to create multiple versions of the library in order to create multiple versions of a program.

Reverting to a previous revision of a program becomes more complex if the revision is not contained in the library.

Allowing multiple revisions in a program library does offer increased

flexibility at the cost of potential confusion to the user if used indiscriminately.

Since allowing multiple versions and revisions in a program library does offer increased flexibility, and since it is not difficult to support and would be optional, the MDS program library was designed to support multiple versions and revisions of programs.

12.4.6 Partially Linked Modules

The concept of partially linked modules is beyond the scope of the Ada language definition and it is not a requirement that a program library support partial links. In fact, extending the program library concept to include partial links introduces a problem which would not otherwise arise. The problem is that the program bodies which are included in a partial link duplicate program bodies which will satisfy the associated program unit specifications - thus requiring a rule to determine which of two or more bodies should be used to satisfy a reference when performing a subsequent link.

However, because of the practical convenience of partial links we feel that the program library should support this feature.

The problem of choosing between duplicate program links is resolved by only selecting a partially linked program body if it is explicitly included in the link structure with an INCLUDE directive.

This approach retains the convenience and efficiency of partial links and still satisfies all normal linker usage.

12.4.7 Non-Ada Programs

Ada provides for interfacing with subprograms written in another language through the INTERFACE pragma in the subprogram specification, the actual body of the subprogram will be written in some other language as specified in the pragma.

An object module conversion tool will be provided which will convert an object module produced by some other language processor into a standard Ada format, the converted module will be added to a program library as the program body of its associated program specification.

Which foreign languages are supported (e.g., FORTRAN) will be determined for each target.

13. COMMAND LANGUAGE INTERPRETER

The editor and the command language interpreter are two parts of the system which are most visible to the user and which tend to arouse a great deal of emotion, both pro and con. The command language is necessary to allow the user to initiate processes. It should facilitate running programs, and not be a deterrent.

There are a number of issues regarding the design of the command language. One of the more important issues is how much the command language should resemble the system's predominant (or only) programming language, in this case, Ada. One environment in which the command language and the programming language are essentially the same is Interlisp [Teitelman81]. It has been suggested that Ada be used as the basis for the command language for the MAPSE [Stoneman80], but it has also been argued that Ada is not well-suited for this [Brender80].

We feel that the nature of Ada and the normal usage of command languages conflict to the extent that it would be unwise to attempt to retain too close a relationship between the command language and Ada. Ada was designed as a language for the implementation of systems for embedded computers. One of the chief aims of the language designers was readability, since many embedded systems have long lives. In contrast, many commands are never viewed by anyone other than their author and the command language interpreter. A large number of commands are only executed once, so ease of entry becomes more important than readability.

Ada tends to be somewhat wordy and has a complex syntax. A command language, particularly one which must be interpreted on a small host, should be simpler. It is worth noting that LISP has a very simple syntax. Thus, it lends itself more to being a command language than does Ada.

An additional problem with using Ada as a command language is that Ada is oriented more toward the use of variables than toward the use of constants, such as character strings or enumeration values. The invocation of a typical system command or user program normally involves passing such items as file names and flags to the programs. In an interactive environment these will most often be constants, except during the execution of a command file. Thus, it becomes inconvenient to have to quote file names, since they occur so often. It should be noted that Interlisp recognizes this problem, also. Commands which are typed in are evaluated using EVALQUOTE which assumes quoting of arguments. On the other hand, during the execution of a LISP program, quotes are not the default.

Another issue is the level of the command language. It is possible to draw an analogy between command languages of various levels and programming languages

of various levels. [Snodgrass80]. At one end of the scale is OS/360 JCL. "As with assembly language, anything is possible in JCL and almost everything is difficult" [Snodgrass80]. Command languages with some basic control constructs are likened to FORTRAN. More advanced command languages are compared with Algol. The language developed by Snodgrass was object oriented and could be likened to programming languages such as SIMULA.

The MDS command language falls toward the lower end of the third class, that which is analogous to Algol. The command language has recursive procedures and scoped variables.

There are a number of systems which have influenced the design of the Microprocessor Development System command language. The most important influence is that of the Unix shell, both directly and through the AIE designs [CSC/SEA81, Intermetrics/MCA81]. Large host systems which had some influence on the design are TOPS [DEC78], TENEX [TENEX], CSTS [CSC74] and Multics [Multics]. IBM JCL was a negative influence, since it is exceedingly complex. Some microprocessor systems which were examined are UCSD Pascal [UCSD] and CP/M [CP/M78, Zaks80].

It should be noted that there are already microprocessor hosted implementations of Unix and Unix like systems [Electronics, April 7, 1981].

The design aims which shaped the command language are as follows:

1. Common operations such as program initiation should be simple.
2. The command language sould be easy to implement in a small amount of space.
3. The language should allow for expansion as hardware costs decrease relative to labor costs.
4. The command language should provide flow-of-control constructs.
5. The syntax should be the same for commands read from the terminal as from a command file.

The following discussion touches on some of the highlights of the command language. A formal definition of the command language syntax is given in the specification of the CLI (Volume 2).

The command language proposed for the MDS may be described as a cross between Ada [Ada Reference] and the UNIX shell language [Bourne78]. The Ada influence is felt most heavily in the control constructs; the UNIX influence in the

program/procedure invocation syntax.

Since it was felt that control constructs were less likely to be entered directly from a terminal (since a user can make a decision as to what to do after the previous program or procedure has been executed), the wordiness of Ada is not really detrimental. On the other hand, it is very common to invoke a program or CLI procedure interactively. Therefore, the commas and parentheses in the Ada syntax for procedure invocation were made optional.

Pipelining and input/output redirection are popular and useful features of the UNIX shell command language. They make it easier to compose more complicated tools from simpler areas. Although these features are not strictly necessary, they are convenient for the user, and have already been implemented on microcomputers [Lycklama78, Cherlin81].

A major design issue for the command language is what to do about declarations. We felt that it was imperative to deviate from Ada here in order to spare the user from having to declare every command language variable. The only declarations which are required or permitted in the command language are those for parameters to command language procedures. The only operand types permitted in the command language are strings and lists of strings. Since literals are expected to be more common than variables, quotes are not required on strings unless they contain any of the delimiters: space, comma or parentheses.

Although the only data types are based on strings, some integer arithmetic operators are provided. Strings which are operands of these operators must be character strings representing integers. (Recall that since quotes are not required, they may be written as simple integers.)

One feature of the command language interpreter which is of particular interest is parameter checking. Rather than requiring each program or tool to check its own parameters for validity, the CLI performs some rudimentary checking itself. Information pertaining to parameters is stored by the linker in the executable object. The CLI makes use of this information to perform validity checking on those parameters which are of type integer, boolean, and file name (a type which is defined in package SYSTEM. Note that string parameters require no checking and that lists of strings are turned into variant records with the CLI supplying the count of the number of strings.

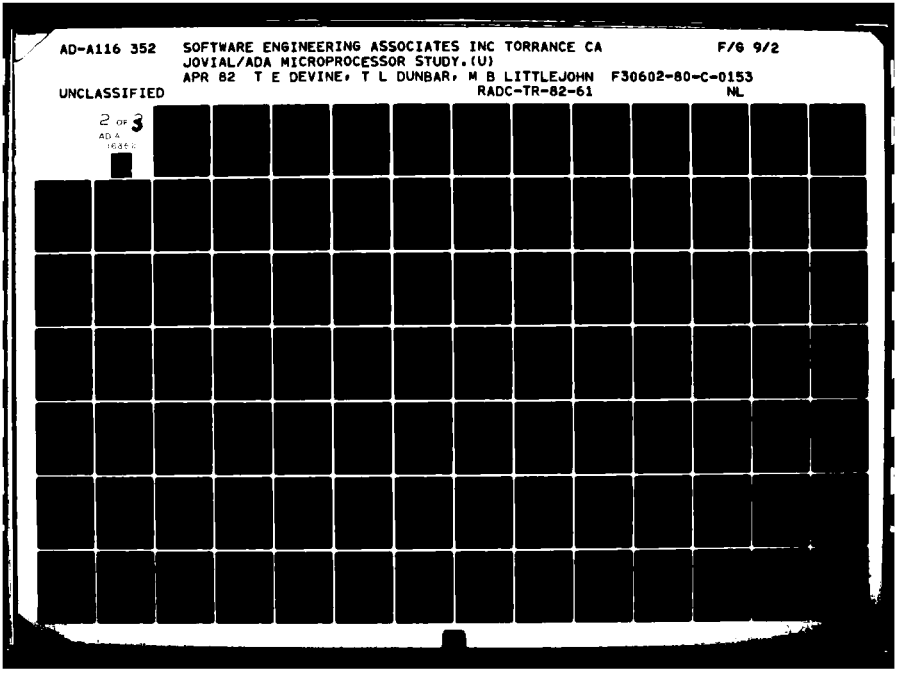
There are user-specifiable search rules which allow a number of partitions to be searched, in order, when a program is executed. Thus, during development it is possible to specify that the most stable, or most experimental versions of programs be used as defaults. In addition, it is possible to specify a particular library to be used when one is not explicitly supplied to a tool.

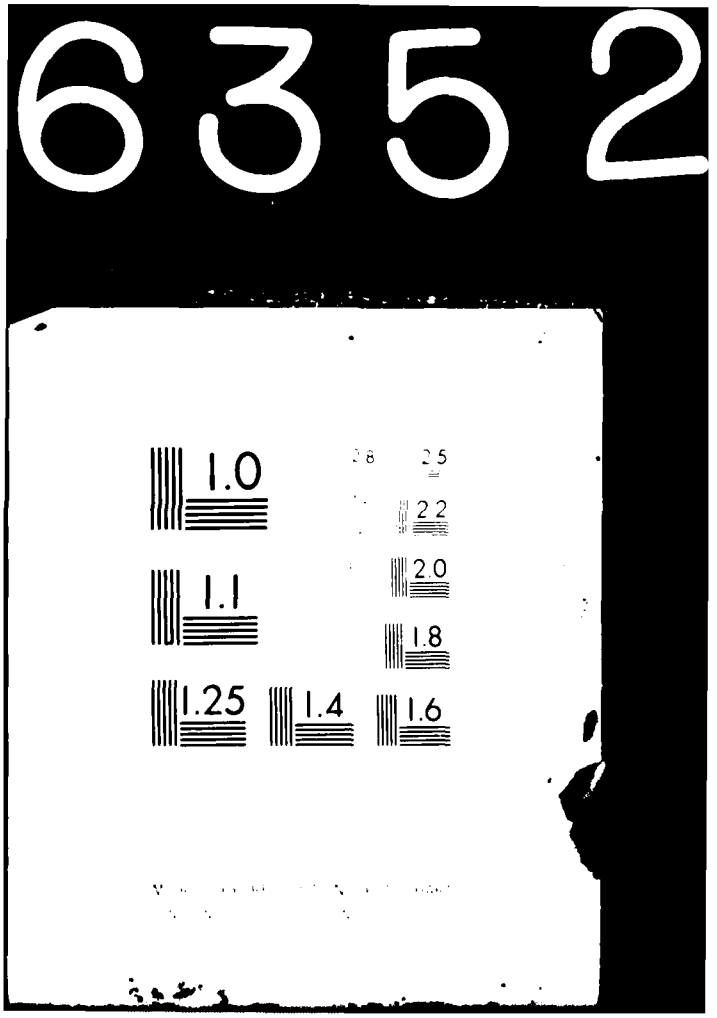
Example:

```
set_default_part *testlib
ada test
link test
test
```

AD-A116 352 SOFTWARE ENGINEERING ASSOCIATES INC TORRANCE CA F/G 9/2
JOVIAL/ADA MICROPROCESSOR STUDY.(U)
APR 82 T E DEVINE, T L DUNBAR, M B LITTLEJOHN F30602-80-C-0153
UNCLASSIFIED RADC-TR-82-61 NL

2 OF 3
AD A
116352




6352
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5

14. ADA COMPILER

The compiler is one of the most important, if not the most important, of the the MDS system tools. Since the system itself will be coded in Ada, the efficiency of the code generated by the compiler will affect the performance of the system.

There are three separate sets of issues which were addressed during the design of the compiler. These categories are:

1. Ada language issues
2. Issues resulting from the size of the host
3. Compiler useability issues

Ada is a large language, containing a number of features whose implementation is complex in and of themselves, but which become even more complicated when they interact with other features. These features include: generics, in line procedures, overloading, separate compilations, tasking and exceptions.

The size of the host system can impose considerable *constraints* on the compiler design. Indeed, if the size constraints are severe enough, meeting these constraints can become the primary consideration. When the 128K goal was selected, it represented a compromise between a minimum hardware configuration and one which would be sufficiently large so that size would not be a major issue.

The 128K size does impose constraints on the size of the compiler and of compiled programs, but these constraints are not unreasonable. It is necessary to break the compiler up into a greater number of memory loads than would be necessary for a compiler which was designed to run on a larger system. It is also necessary to sacrifice some conveniences, and processing time as well, in order to make the individual phases smaller.

14.1 Symbol Table

One of the most important factors with respect to the region required for a compilation is symbol table strategy. Having a resident symbol table simplifies the logic of the compiler, since it is not necessary to have a mechanism for bringing in, or writing out, the symbol table or parts thereof. However, this can require large amounts of space. Measurements made with the SEA J73 compiler indicate that modules in the 4000 line range require around 60,000 bytes of symbol table space, exclusive of that required for names and defines. One module of approximately 5800 lines required over 90,000 bytes for the symbol table (see Appendix D).

An alternative strategy would be to have a virtual symbol table. This would be a good approach on a system which provided hardware virtual memory, provided that references to the symbol table exhibited locality. However, at the present time, microprocessors do not provide sufficient hardware to allow this to be done. This situation is expected to change soon, however, since Intel, Zilog, and Motorola are developing chips which will permit virtual memory [Callahan81, Johnson81].

It would, of course, be possible to simulate virtual memory in software. This could be done most easily in an interpretive system. Accordingly, the compiler could be coded as if the symbol table were resident. A virtual memory facility could be provided in a native-code system but this would require special subroutine calls prior to symbol table references and after symbol table entry modifications. In such a scheme a pointer would be obtained by calling one of the virtual memory routines, which would return a pointer if the symbol table entry was already in memory, or would cause the entry to be brought in, as necessary. An approach which follows this method is described in [Intermetrics81].

The software virtual memory approaches are conceptually clean but could cause a rather severe degradation of compiler performance, since every new reference to a symbol table entry would require a subroutine call. In a hardware virtual memory system, the cost of memory (in terms of execution time) shows up only when a page fault is generated.

We rejected the virtual memory approaches for two reasons. First there is not, at the present time, hardware which will permit this to be done. While such hardware is expected to be available in the near future (and certainly by 1985), assuming hardware virtual memory would preclude the use of any system which did not provide this feature. Second, although, software virtual memory is certainly implementable at the present time we felt that its effect on compiler performance would be unacceptable.

The approach chosen was to flush those portions of the symbol tables which become invisible due to the scope rules. This approach works best in a one pass compiler, because when a scope is flushed, there is no reason to bring back that portion of the symbol table again. Since the compiler design presented here uses multiple passes, it is necessary to bring scopes back in when they are entered in subsequent phases.

The UCSD Pascal is an example of a one-pass compiler which uses such an approach. At any given time, the only information in the symbol table is that which is visible from the scope currently being processed. The symbol table is built on the heap. At the start of a scope, the heap top is marked. At scope exit, the heap space down to the mark is released.

Because these are some differences in the scope rules of Ada and Pascal (primarily due to packages and overloading), the same method cannot be used directly. However, it is possible to flush data which will not be needed until the next phase. Since a multi-phase implementation is envisioned, it is necessary to insure that any symbol table entries which are modified are written out so that information is not lost between phases.

Another major design decision is where to place the symbol table. There are a number of arguments for placing it in the heap. This allows entries in the symbol table to be of variable size, and allows for the expansion of the symbol table to the maximum size of the heap.

On the other hand, there are some serious disadvantages to using the heap for the symbol table. Heap pointers are inherently larger than entry indices. Thus, symbol table entries which contain pointers are larger than they would be if they only contained indices. This increases symbol table size, which is a problem on a small system.

There is also a cost associated with flushing the symbol table at the end of a scope. There are two problems here. The first is that, unless some assumptions can be made about the heap allocation mechanism, it is not possible to write out the symbol table in blocks. This assumption would require that heap space is allocated linearly. While this could be done, it would detract from the portability of the compiler. Another problem is that, unless a conversion is performed, heap space pointers would be written out and read back in. This would cause dangling pointer problems, unless it could be guaranteed that the symbol table fragment could be read in to the exact locations from which it was written and furthermore, that none of that space had been reallocated for another purpose in the meantime.

We felt that for a multi-pass compiler the disadvantages of using the heap outweighed the advantages. For this reason the compiler design incorporates a symbol table which is implemented as an array of records, rather than in the heap. This allows the use of short pointers (16 bit indices, for example) and allows segments of the symbol table to be read or written as blocks.

There are some disadvantages to this approach also. The most important disadvantage is with respect to symbol table expansion. It is possible to obtain space for the symbol table based on the amount of memory available to the compiler. It would be difficult, however, to expand this table once space had been obtained without resorting to trickery, such as turning off subscript range checking and using subscripts outside the range of the array. This, however, can be quite dangerous. The other disadvantage is that the size of the symbol table has an absolute maximum of number of entries based on index size. If index fields are declared to be a certain size, say one 16 bit word, then it is only possible to address 64K entries in the symbol table,

regardless of how much space is obtained. This does, however, seem to be a reasonable limit for the symbol table.

There is also somewhat of a problem with respect to the size of symbol table entries. If entries are of variable lengths, space will be wasted if entries are allocated based on the maximum size of an entry. On the other hand, if arrays are implemented in such a way as to allow for variable length entries (such as by means of a hidden pointer array), array references will not be as efficient. The approach which will be used here to get around this problem will be to use fixed length entries and to use auxiliary entries for those classes of entries which require more information than may be contained in the entries.

Another approach which was considered but rejected is the distributed symbol table approach, such as that used in Aida [Persch80]. In this approach, there is no symbol table, per se. The problem with this sort of approach is that information cannot easily be added to the symbol entries after resolution has been performed. Of particuliar concern here is information such as the use counts and reference contexts of operands (results), which can be helpful in generating better code.

14.2 Compiler Structure

Another major design decision is whether to pass the source program by the compiler or to pass the compiler by the source program. Typically, the program is passed by the compiler. Each phase of the compiler processes the entire source program (possibly in an intermediate representation) before the next phase is loaded. In such an implementation each phase reads some form of the program and writes a modified version.

An alternative implementation is to pass the compiler by the source program. In such an implementation, the source program remains in memory and compiler phases are loaded in sequence. This allows a large number of small compiler phases to act upon the program. Such a scheme is more attractive for a language such as FORTRAN which has only one scope per compilation unit. For a language such as Ada, this approach is less attractive, because the alternatives are to hold all scopes in memory at once and load the compiler phases just once, or to load each phase for each scope in the source program.

14.3 Useability

There are a number of issues relating to compiler useability which conflict with some of the constraints discussed previously. In particular, providing the features which make a compiler useful for the development of real systems

can require substantially more room than would be required for a "toy" compiler.

Such features as cross-reference listings, and meaningful diagnostics make a compiler more useful for serious program development. Object code quality is another important requirement for a production compiler. For applications such as embedded computer systems (for which Ada was designed), the code generated by the compiler must not be less efficient than assembly code by too large a margin.

A production compiler must be able to handle modules of a reasonably large size. Ideally, the compiler should be able to handle any size module. However, this goal conflicts with practicality considerations. Since the MDS Ada compiler will run on relatively small hosts, we feel that the size restrictions for compiled programs may be somewhat more stringent than for large-host compilers (cf. AIE) but must not be overly restrictive.

The design which is described here should be able to handle modules which are several thousand lines long, provided that the compilation units are broken up into reasonable-sized routines.

Compiler speed is another important consideration. While it is certainly desirable that the compiler use a minimum of CPU time, a better measure of compiler speed from a user point of view is wall-clock time, since that gives a better indication as to how long the user is delayed, waiting for a compilation to finish.

One of the major issues with respect to compiler design for a small system is whether interpretation should be used. Code size can be significantly reduced, particularly on a host which does not possess a powerful instruction set [UCSD78, Donegan78]. For some of the older 8-bit microprocessors interpretation is very attractive, since subroutines would be required even for such simple operations as multiplication. However, the newer 16-bit micros have instruction sets which are sufficiently powerful so that interpretation is by no means a necessity, and would cause a substantial reduction in speed. For this reason we favor a native code implementation of the compiler.

One of the basic design goals was that the compiler should do a good job on those things which are most common. Rare cases would be handled well, if possible, but the rare cases would receive less emphasis.

One example of this is in generating code for expressions. There are any number of algorithms which describe how to generate code for complex expressions using the minimum number of registers. On the other hand studies [Knuth71, Bloom74] have shown that the vast majority of expressions which

appear in real programs are quite simple. For this reason our emphasis is on generating good code for simple expressions, rather than for more complex ones.

Since resources are relatively scarce on a microprocessor, it is necessary to be rather conservative for an initial implementation. It is likely that some compiler "tuning" will be necessary, since Ada is a new language and statistics on its usage are not yet available. It is possible to make educated guesses as to the effect of new programming techniques such as the use of abstract data types on the use of the language, but statistics of actual usage would be more useful. The design described here accomodates such tuning, since it is relatively easy to add additional phases as necessary.

There are a number of features which are new to Ada with respect to J73. How these features are used will have an effect on compiler performance and may provide opportunities for obtaining significant improvements by handling certain common cases effectively. Some of these features are overloading, generics, packages, operations on aggregates (including the ability to define functions which return aggregates and tasking). It is likely that most programmers will use relatively small subsets of these facilities heavily and will largely ignore other subsets. This is, in part, because programmers are creatures of habit and partially because programs which use language features which are in the mainstream are safer, since commonly used features are better understood, and because the compilers are better checked out with respect to these features.

14.4 Parsing Technique

There are a number of candidates for parsing technique [Aho72]. We have selected LALR(1) on the grounds that it is efficient as well as powerful enough to be able to handle Ada with relatively few modifications to the grammar. A LALR(1) grammar for revised Ada already exists [Persch81].

14.5 Optimization

The design of the optimization phase(s) of the compiler is one of the more critical areas with respect to the useability of the compiler as a development tool for embedded computer systems. Since embedded computers typically have limited resources, it is necessary that the code generated by the compiler be both compact and fast. If compiler code is not efficient enough, the developers of embedded systems may be tempted to resort to the use of assembly language.

Optimization, however can be potentially costly during compilation, both in

terms of time and space. For this reason, optimizations must be selected on the basis of a cost/benefits analysis, rather than because they sound attractive. As was discussed previously, it is our intent to do a good job for the high-payoff areas and to place less emphasis on the rarer cases.

14.6 Local Code Generation

Perhaps the most important area for the generation of good code is local code generation. This is not strictly the province of the optimizer, but rather of the code generator. In order for the code generator to generate good code for the myriad special cases which occur, the generation routines must be provided with sufficient information to make intelligent choices of code sequences. Code must be generated in context. It is necessary to know how a result is being used in order to generate good code. One common example of this is boolean expressions in conditional contexts. There is no need to generate a storable value for the boolean expression -- it is sufficient to set a condition code, for example. Another example of where context information can be used is in the generation of increment or decrement sequences rather than load, add (subtract), and store sequences.

14.7 Calling Sequences

One place where a high payoff may be obtained, but which is often neglected, is subroutine calling conventions. Often the compiler writer is limited with respect to the types of sequences which can be used, due to previously existing system conventions. Also, it is often desirable to provide traceback information in the absence of a source-level debugger. The MDS system can assume the presence of such a debugger, so it is not necessary for traceback information to be included in the resident object code -- the debugger can obtain the information from the non-resident debugging tables.

The Ada program library and recompilation rules make it possible for the compiler to use "negotiated" calling sequences rather than standard calling sequences, by recording the appropriate information in the library. For example, it may be desirable for a given subroutine to receive one of its parameters in a certain register. The compiler could generate code for the subroutine, taking advantage of this fact, and record the parameter conventions in the library as part of the subroutine specification. When a caller is compiled, this information would be available. This would enable consistent code to be generated. If the subroutine were recompiled with a (compiler-generated) change to the calling convention, this would be handled in the same way as if a (user-generated) change had been made to the specification.

14.8 Addressing

Addressing optimizations are another source of substantial savings, primarily with respect to code size. Many architectures, including most of the more recent microprocessors, have short and long forms for addresses. Often the short forms are half the size of the long forms. Most machines of recent vintage offer two forms of addressng for branches. Many also offer different address lengths for operand references as well.

While the basic ideas behind the two types of optimizations are the same, there are some significant differences. Most often the branch optimization is based on the use of a relative branch, rather than an absolute jump. In the usual case, the branch is relative to the program counter. For references to data, the choices are usually between a reference relative to some register which contains a base address, and the use of a full address. Another possible alternative for some architectures is to use a short pointer to a long address and to use indirect addressing.

It is not necessary for the compiler to allocate the program counter, since it must exist, anyway. For data reference, however, if cover registers are to be used, code must be generated to load a register. This may be done by convention (as is often done with the local frame pointer) or as the result of an optimization decision. While the compiler has the freedom to reorder code to create a higher proportion of short to long references for code, payoff would likely be too small to justify such an effort. On the other hand, the reordering of data is likely to produce results which do justify the effort expended. Data can be allocated so as to allow more commonly referenced data to be accessed using short addresses.

14.9 Exception Checking

Generating worst-case code for subscript checking and range checking can lead to enormous increases in both the size and execution times for programs in which checking is not suppressed. Increases of over 100% in execution time have been observed in this study and elsewhere [Welsh78]. The difference is less for interpreted code than for compiled code, but it is still substantial. For a bubble sort test case the increase in execution time was 100% while for interpreted code it was 28%. For quicksort the differences were even more dramatic — 282% and 35%.

These results are somewhat extreme, since the test cases in question have a larger proportion of array references than would be found in an average program. However, even for a typical program, the cost of unoptimized checking would be too great to ignore in an embedded computer environment. While Ada allows checking to be suppressed via the suppress pragma, it is not

desirable to suppress checking if it can be avoided, since the elimination of checking can allow bugs to manifest themselves in ways which conceal the true nature of the problem.

Since the MDS compiler should encourage good programming practice (or at least not discourage it), it is necessary to include checking optimizations in the compiler. These optimizations are based on [Welsh78] which is, in turn, based on [Suzuki77]. An extension must be made, since Ada permits arrays whose bounds are unknown until execution time, while Pascal arrays all have compiler-time bounds. [Fisher77] discusses run-time checking for Pascal; a number of problems discussed there have been remedied in the design of Ada.

The key to safely eliminating run-time checks is the use of range declarations and flow to know when it is impossible for an exception to be raised. Some cases are trivial. If the right side of an assignment is of the same type as the left side, it is not necessary to generate a range check. Other cases are more complex. [Welsh78] shows examples of using the range of a control variable of a for statement to eliminate checks within the scope of the for statement.

14.10 Register Allocation and Common Subexpression Elimination

Register allocation and common subexpression elimination are the closely-related optimizations which are also important. Register allocation may be viewed as a special case of the more general storage allocation problem. Although registers must be allocated even in the absence of common subexpression elimination, the problem is much more difficult when there are common subexpressions. There are a number of approaches which can be taken with regard to register allocation. For straight-line code optimal solutions are known, provided common subexpressions are ignored. While these methods are of theoretical interest, they are not really practical for the MDS compiler; complex expressions are relatively uncommon [Knuth71] but common subexpressions occur frequently [Lunde77].

14.11 Dead Code Elimination

Dead code elimination is an important optimization in Ada, since it is the means for achieving conditional compilation. Dead code may also arise from a call of a inline procedure with a constant parameter, or may be created due to other optimizations. It may also arise if the code generator produces suboptimal code sequences with the understanding that they will be cleaned up by a later optimizing pass. For example, it may be simpler for the code generator to generate a branch over the else part of an if, even where the then part is a goto.

14.12 Constant Arithmetic

Constant arithmetic is another important optimization, particularly in light of the fact that Ada allows programs to be parameterized by means of constant declarations. (It should be noted that constant arithmetic refers to compile-time constants, while constant declarations in Ada may declare objects which are compile-time constant or perhaps constant only for a particular invocation.)

It is desirable from the standpoint of program readability and adaptability to encourage the use of named constants, rather than literals. In order to do this, the compiler should not impose an execution-time penalty when such constants are used.

Constant expressions may also arise due to the use of attributes or inline procedures. It may be impossible for the user to evaluate these expressions and code their values in the source program. Therefore, the job must be done by the compiler.

There are several places where constant arithmetic must be performed. The language requires that certain expressions be evaluated at compile time. These evaluations must be performed in the front end of the compiler. Other expressions may not be known to be constant until optimization has been performed. Examples of such expressions are those which become constant due to folding.

14.13 Flow Analysis

There are a number of possible choices for a flow analysis algorithm. Various methods for global flow analysis are described in the literature [Kildall73, Cocke70, Kennedy75, Wulf75, Loveman76]. It would also be possible to ignore the problems of global flow analysis and worry about data flow within basic blocks or on a regional basis. The method chosen is a compromise between the fastest methods (which do not allow certain optimizations to be performed) and the most general (which can require large amounts of space and time during compilation).

Ignoring data flow in loops (or more correctly, making worst case assumptions) prevents loop optimizations such as the removal of loop invariant expressions from within loops. Since large amounts of execution time tends to be spent in small areas of a program (loops) [Knuth71], (inner) loop optimizations can have a substantial influence on execution speed.

On the other hand, while a flow analysis method such as *P*-graphing [Loveman75] is completely general (in the sense that exactly which definitions can reach

which uses for arbitrary program flow), the algorithms require large amounts of time and space. Restricting program flow graphs to well-structured programs allows more efficient methods to be used [Wulf75, Geschke72], but at the cost of some generality. Note that "spaghetti" code can still be compiled -- it just will not be optimized to as great an extent.

We feel that this is a reasonable choice for several reasons. First, Ada provides control constructs which are sufficiently powerful so as to reduce the need for goto statements. Conditional statements are somewhat more general than in languages such as Algol, J73 or Pascal, with the addition of "or else" and "and then" as well as "elsif". Ada has a case statement, also. The loop statement is sufficiently powerful so that most loops can be coded as formal loop statements. The exit statement serves to eliminate the need for a relatively common class of gotos.

It should be possible to extend the algorithms to handle special classes of gotos (such as forward branches) at a moderate cost, should this become necessary. The fact that Ada restricts the possible targets of gotos (e.g. not out of a subroutine, nor into a loop) would make such an extension easier. It is not anticipated that this will become a critical addition to the compiler. Therefore, it is proposed that the flow analysis be done in the more restricted form.

14.14 Compiler Structure

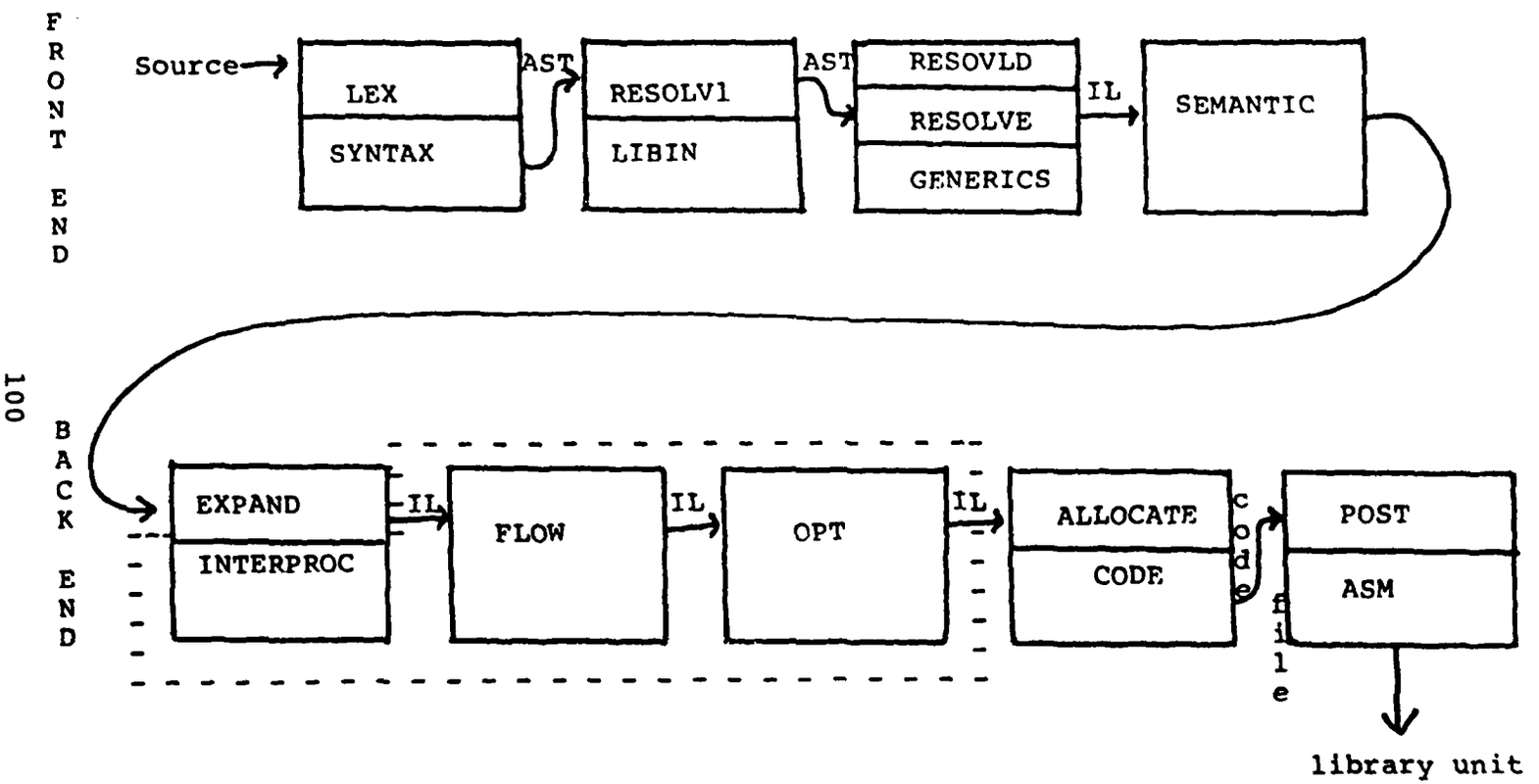
The compiler is divided into a machine-independent front end and a machine-dependent back end. There are 16 logical phases which are packaged into 9 memory loads as shown in the diagram below. The roles of the phases are as follows:

LEX	performs lexical analysis
SYNTAX	performs syntactic analysis
RESOLV1	performs preliminary resolution - builds a list of those names which may potentially require resolution by package data
LIBIN	brings in specifications from the library
RESOLVD	resolves names in declarations
RESOLVE	resolves names in executable statements
GENERICS	instantiates generics. Note: RESOLVD, RESOLVE, and GENERICS operate in parallel
SEMANTIC	performs the remaining static semantic analysis
EXPAND	expands the IL in a target-dependent manner
INTERPROC	performs interprocedural analysis
FLOW	performs flow analysis
OPT	performs optimizations
ALLOCATE	allocates variables

CODE	generates code sequences
POST	performs post-code-generation optimizations (including peephole optimization)
ASM	generates relocatable object or assembler input

Although they are considered part of the back end, INTERPROC, FLOW, and OPT are machine-independent. EXPAND inserts machine-dependent information into the IL so that these three phases can perform optimizations appropriate to the target machine, without having knowledge of the target built into them. The dashed lines indicate the optional (optimizer) phases.

Compiler Structure



14.15 Compiler Use

The compiler requires the specification of a source program and library, although the library may be defaulted. Facilities are provided for selection of default versions and revisions of library units, but a particular version or revision of a unit may be specified explicitly. The compiler generates no listings; these are provided by separate tools.

15. LINKER

The linker combines relocatables to form executable programs or partially linked modules. It facilitates the development of programs by different programmers or groups of programmers. It differs from more traditional linkers in that it must support compilation order checking. In addition, the linker makes use of the attributes of program library members in order to make the linking process more automatic in situations where explicit version/revision selection by the user is unnecessary.

The design goals of the linker are as follows:

1. It must support all Ada requirements, in particular, compilation order checking.
2. It shall support the requirements of small to medium scale program development efforts.
3. It shall help to automate the recompilation process, in the event that compilation order errors are detected, but allow the user to maintain sufficient control.
4. It shall be easy to use.
5. It shall be efficient.

The linker design is based largely on the CSC/SEA linker design for the AIE [CSC Linker81]. Most of the changes from that document are editorial in nature. The major substantive change is the addition of a capability to generate a recompilation order list suitable for driving the recompilations. The deletion of unused procedures from packages is now stated explicitly.

This chapter provides a general overview of the Linker and describes the design tradeoffs during the Linker preliminary design. The design itself appears in an accompanying volume.

15.1 Interfaces

Both the Ada compiler and the Linker will make use of a common relocatable object formatting package for formatting the respective output object.

Some targets may require that a special target-dependent load object format be produced. In this case a tool would be provided to transform the target-independent load object to the target specific format.

All programs developed under the MDS must be linked before they can be executed. Under user direction, either from the Link command or from a Link directive text object, the Linker reads program library and member relocatable objects produced by the Ada compiler, or produced by the Linker itself from a

prior partial link, and creates a single relocatable load object. The Linker will interface with the database to access and update program libraries and relocatable objects. The linker records a description of the parameters of the load object so that the CLI can perform type checking on parameters whenever a load object is executed through the CLI invocation mechanism.

15.2 Capabilities

The explicit Linker requirements specified in the SOW, STONEMAN and the Ada Reference Manual are minimal.

The common Linker functions are an implied requirement of the Ada language because of the need to create programs from separate, independently compiled compilation units. These Linker functions include the support of multiple location counters, the resolution of external references and the relocation of address references.

In addition, the concept of a program library is defined in Ada to ensure that a program consisting of several independent compilation units will have the same degree of type safety as the same program submitted as a single compilation. The Compiler largely supports this compatibility requirement; however, the Linker further satisfies this requirement by validating the actual order of compilation of the objects in a link.

The fact that there are no external references in the conventional sense in Ada means that it is not necessary to do resolution for user-defined names. At compile time it is known which library unit contains the referenced object or routine, since definitions can only be obtained from other units by means of "with" and "separate" clauses. Thus, the compiler is able to inform the linker which unit must be used to satisfy references. In the case that multiple versions and revisions are allowed (as in the MDS), there is a set of eligible library units, and there must be a means of selecting the correct one. This is done in this design by allowing default versions and revisions to be specified, and by allowing the user to instruct the linker to include a specific version and revision of a particular unit. Explicit versions and revisions need only be supplied to override the defaults; they are not otherwise required.

Further, the Ada language specifies a required order of elaboration of library units included in a program. The Linker satisfies this requirement by creating an elaboration procedure for each load segment; this procedure performs the required library unit elaboration in the proper order at execution.

In addition, the definition of exception handling in [Ada Reference] implies

that the linker perform a sort of "relocation" on exception codes.

Beyond these limited requirements the functions supplied by the MDS Linker are those that are required in a useful user-oriented program development system and that are available in some form in many commercial operating systems.

These functions are outlined below.

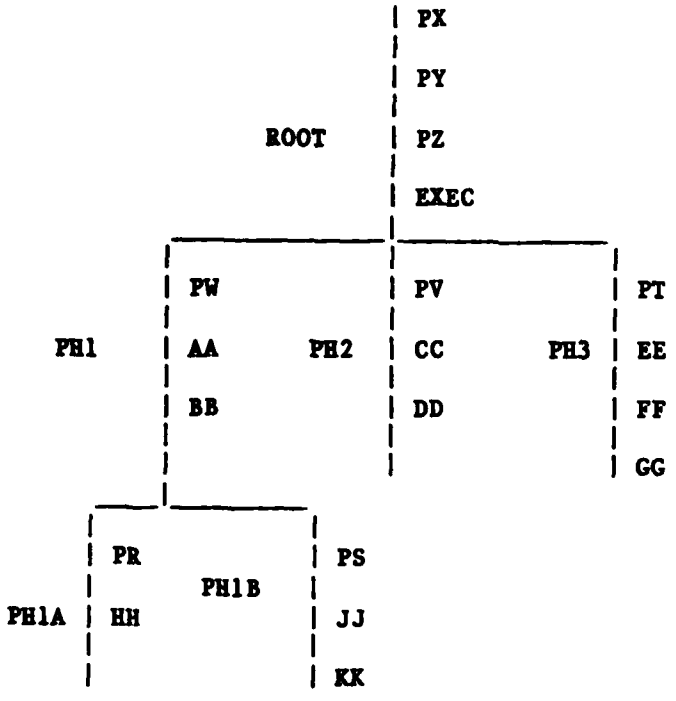
Partial Linking

To facilitate construction of large programs, it must be possible to link portions of the program independently, form them into partial link objects, and present them as input into a larger link activity. This idea is analogous to independent subprogram compilation. To support this, the Linker will construct linked objects in relocatable object format so that the Linker can accept its output as input to a subsequent link. Furthermore, the user is able to specify in partial linking those external symbol definitions within the link that are to be retained as external symbols for resolution of references *from outside* the link. This approach results in a far more efficient use of the system resources by not requiring a total relink when only a few compilation units within an overlay segment are recompiled.

Multilevel Overlay Structure

Many linkers support only simple tree-structure overlays. This is clearly not sufficient for many large program organization efforts such as command/control systems or compilers. The MDS Linker supports this requirement by allowing the specification of a multilevel overlay program structure through simple linker directives.

As an example the following diagram pictures a simple two-level overlay structure. Names to the left of the overlay legs are used to represent segments, names to the right represent library units.



The link directives to build such a program are shown below:

```
ROOT    SEGMENT
        INCLUDE   PX, PY, PZ, EXEC
PH1     SEGMENT
        INCLUDE   PW, AA, BB
PH1A    SEGMENT
        INCLUDE   PR, HH
PH1B    SEGMENT   PH1A -- overlay PH1A
        INCLUDE   PS, JJ, KK
PH2     SEGMENT   PH1 -- overlay on PH1
        INCLUDE   PV, CC, DD
PH3     SEGMENT   PH1 -- another overlay on PH1
        INCLUDE   PT, EE, FF, GG
        END
```

Following the execution of these link directives, the following library units will have been linked in the following order; PX, PY, PZ, EXEC, PW, AA, BB, PR, HH, PS, JJ, KK, PV, CC, DD, PT, EE, FF, GG.

Automatic Segment Fetching

In a program with an overlay structure it is necessary that the overlay segments be loaded before they can be referenced. This is supported by the MDS Linker as follows. Every call to a procedure located in a lower level overlay segment is replaced by an indirect call to a system routine that will check to see if the referenced segment is loaded. If the segment is not loaded, the referenced segment and any unloaded higher level segments will be loaded before executing the procedure call. Once the procedure (or the segment containing the procedure) has been loaded, the indirect reference will be changed to point directly to the subject procedure.

This approach allows the program overlay structure to be modified without requiring any source changes or any recompilations - only a new link is needed.

Heap and Stack Space

For each target system a suitable default size and location will be defined for the heap and stack space. Where the default is not adequate, the user may specify the size or the allocation of heap and stack space and whether or not the heap and stack space should share a common area or should be controlled separately.

Library Unit Elaboration

The Ada language has specific requirements concerning the order of elaboration of library units in the execution of a program.

The Linker must ensure that elaboration is performed for all library units in the program in the proper order. This is accomplished as follows.

Each compilation unit that requires specific elaboration has a callable elaboration prologue generated as a procedure by the compiler in a standard location in the code section of the object. For each object included in a load segment, the Linker finds its elaboration prologues (if any) and includes a call to it in an elaboration procedure for that segment at the start of the code section for the segment.

The elaboration procedure consists of a set of calls, in the proper order, to the elaboration prologues in the segment. When a segment is loaded, its elaboration procedure can be executed to perform the elaboration of every library unit in the segment.

Stub Generation

The Ada Compiler produces a program library unit specification for every "is separate" procedure that identifies the procedure as being a stub. When the Linker includes such a stub procedure, it is replaced with a dummy procedure that simply returns. Optionally, the dummy stub procedure will raise a SOURCE_ERROR exception.

Compilation Order Validation

The Linker will perform the final verification of the Ada requirements for proper compilation order. All violations will be reported to the user although they will not abort the link. A recompilation order list, suitable for driving recompilations, will be produced.

Module Promotion

Where an object is referenced in two or more overlay segments of a program, that object will be promoted to a higher level segment that is common to those overlay segments and no others.

If an object is explicitly included in a segment with the INCLUDE directive, that object will not be promoted out of the segment. This allows the user total control over the program structure without relying on the linker defaults for implicit library unit inclusion.

Boundary Alignment Module Placement

The user will be able to specify boundary alignment for any externally relocatable element of the link such as an object or location counter. The boundary alignment could be expressed as an absolute address or as some function of the next available location, such as double-word alignment, next byte, or next page. The user will be able to place objects in the linked program in a particuliar order or allow the Linker to choose the order.

Linker Listings

The Linker itself will not produce listings. Rather, there will be separate tools to produce user-oriented map and concordance listings from the debugging tables. The map will show the allocation and the attributes of the various program location counters and entry points for each object and segment in the linked program. The concordance listing will show the referencing library

unit name, the referenced library unit name, and the referenced element name for every external reference from a compilation unit. The order in which these names are applied in the sort may be user-specified.

16. DEBUGGER

The debugger provides facilities for program debugging at the source level. The debugger design presented here is based on the CSC/SEA Debugger for the AIE [CSC Debugger]. The primary change is the insertion of the definition of an "address". This definition appeared only in the Interim Technical Report [CSC ITR]. Its omission led a number of reviewers to conclude that the debugger required machine addresses and did not allow for source-level debugging. In addition, an attempt has been made to make the syntax less cryptic.

The MDS Debugger supports program developers by providing comprehensive symbolic interactions with an executing Ada program. Facilities supported by the Debugger are Ada data object examination and modification; controlled execution in the form of single stepping, breakpointing and/or interrupting; and monitoring and tracing of program execution. The Debugger performs its function by direct execution of the subject process controlled by instruction implants rather than by simulation.

The principles that influenced the design of the Debugger are:

1. The Debugger must allow the programmer to debug at the level the program was developed, namely at the Ada source level.
2. The Debugger must support the checkout of operational, fully-optimized programs.
3. The Debugger must be efficient and not require excessive memory or processor resource usage.
4. The user need not anticipate the existence of program errors and their location in order to correct them effectively.
5. The Debugger must support the checkout of programs that utilize the full Ada language including Ada tasking.
6. The Debugger must support the checkout of programs compiled for other target computers and their checkout in both the host environment and through use of simulators in the host environment.
7. The Debugger must support the location and correction of program errors and the performance tuning and quality assurance aspects of program development.

The following section provides a general overview of the Debugger interfaces within the MAPSE and describes the design tradeoffs performed during the Debugger preliminary design.

The Debugger is a functional part of the CLI and is invoked when a debugging command is encountered in a CLI command stream. The Debugger operates as part of the CLI process and permits a user to control and monitor a child process of the CLI by interacting with the Debugger through use of the Debugger

directives.

The Debugger permits referencing of program identifiers through the use of Ada Debugging Tables (ADTs) produced by the Compiler and included by the Linker in the load object.

16.1 Relation to MDS

The debugger forms a part of the CLI. Thus, whenever debugging is in progress, all CLI facilities are available. The debugger is intended to operate in an address space separate from the program being tested. Of course this can only be done on hosts which provide distinct address spaces.

16.2 Capabilities

The Debugger provides the user with the interactive facility to discover and correct Ada program errors. The functions supported by the Debugger are requested by directives entered through the standard input file (normally the user's terminal). The Debugger controls the execution of the subject process by entering instructions into the user's program as a result of user requests. This debug process and the Debugger operate synchronously — with execution of the debug process proceeding until an implant causes the Debugger to take control. A facility exists to allow the debugger to gain control at the point where an exception is raised in the test program, before an attempt is made to find the test program's exception handler. This allows the user to examine dynamic data before it is deallocated due to the invocation of the exception handler. When the Debugger is in control, the user may examine or set program variables, make procedure calls, enable execution traces, etc., and then resume execution of the process.

Ada Source Level Debugging

The Debugger directives and the Ada name referencing are expressed in Ada syntax. All values whether expressed in a directive or displayed by the Debugger will be represented according to type using Ada literal syntax.

Although the intent of the Debugger is oriented toward checkout to the extent possible at the Ada Language level, circumstances are expected to arise where machine-level observation is required. The Debugger supports this by permitting access to individual instructions, actual addresses, single stepping, instruction tracing and dumps. Even at this level, the Debugger attempts to allow references and display values in as symbolic a manner as possible.

Debugging Optimized Operational Programs

Although the Compiler supports an option to suppress optimization, thus ensuring the user of strict correlation of the source program to the executing program, the Compiler and Debugger have been designed to permit the checkout and correction of operational, optimized programs. The Compiler produces detailed messages that describe the optimizations performed. The ADTs also include descriptions of the optimizations performed with each statement. In particular, the locations assigned to source and compiler-generated variables by the allocation are recorded in the ADTs. The Debugger will then be able to alert the user when debugging requests may be affected by optimizations.

Resource Use

The ADTs that provide the Ada names and attributes are maintained in the load object and do not occupy any memory until an Ada name is referenced. Only small portions of the ADTs are required at a time, thus the memory impact of symbolic name referencing is minimized.

No hooks are required in the code to support debugging. This eliminates the problem of significant code expansion often associated with debugging.

To minimize performance impact on a program being debugged, direct execution is utilized rather than interpretation. Execution is controlled by implanting breakpoints into the program as required. Furthermore, the Debugger operates as a separate process to prevent the results of the program being debugged from being affected unexpectedly.

Error Anticipation

Since error locations cannot be anticipated, the Debugger has been oriented to debugging without hooks and without special preparation of the program for execution. An operating program may be interrupted and debugged with the same facilities as if it had been loaded and started by the Debugger.

Full Language Support

All language features are supported by the Debugger through directives; however, the full Ada expression and statement capability is not provided within the Debugger directives to eliminate the need for full interpretation and the associated debugging performance penalty. The user may interface with and monitor Ada tasks. All Ada types are supported, all Ada program objects may be referenced and examined, and most Ada operators may be used in Debugger

expressions.

Target Program Checkout

The Debugger is designed to permit checkout of programs compiled for target computers being simulated on the host development computer as well as programs being directly executed on the host. The Debugger interfaces are fully standarized; the fact that the program is executed through simulation is transparent to the Debugger.

The Debugger also provides a mechanism to perform data recording and environment simulation through the record directive. By the use of minimal scaffolding and this record function, both static checkout and post data reduction facilities are provided.

Timing and QA Considerations

An often neglected development requirement is an automated mechanism to support program validation and refinement. Two features have been included as an integral part of the Debugger to support this need. These features allow the user to time individual program regions and count region entrances thereby providing performance tuning data as well as ensuring that all program paths have been exercised. The data required to support this facility is a fallout of the compiler optimizer.

16.3 Sample Debugging Session

The following Ada program and debugging session illustrates the use of the directives.

```
 1  package trig is
 2  type deg is integer range 0 .. 360;
 3  type sc is float range -1.0 .. 1.0;
 4  function sin(d:in deg) return sc;
 5  pi: constant := 3.14159;
 6  end
 7
 8  package body trig is
 9
10  function sin(d:in deg) return sc is
11      i: integer range 1..20;
12      limit: integer := 2;
13      factor: float := pi*float(2)/180.0;
```

```
14      term: float := 1.0;
15      sin: float := 0.0;
16  begin -- use Taylor series to compute sine
17  series: while i in 1 .. limit loop
18          term := term*factor/float(i);
19          if i mod 2 = 0 then
20             term := -term;
21          else sin := sin + term;
22          end if;
23     return sin;
25  end sin;
26  end trig;
```

The directives below represent a sample debugging session (the comments on the right are editorial notes). Debugger responses are underlined.

```
load trig                          --loads package trig
go                                 --elaborates package
use trig                           --default name qualifier
sin (30) ?                         --compute expression
= 0.52359                          --not very precise
insert trig.19; [i ?]              --sin.i when encountered
insert #1                          --1st insert
sin (30) ?                         --try again
trig.sin.i=1                       --1st loop iteration
trig.sin.i=2                       --2nd loop iteration
= 0.52359                          --still wrong of course
insert sin.series; [limit:=20]     --series should have 20 terms
insert #2                          --2nd insert
cancel 1                           --disable 1st insert
sin (30) ?                         --one more time
=0.50001                           --much better
quit                               --done, go correct source
```

16.4 Debugger Directives

The Debugger directives use a simple syntax with Ada symbols for Ada program names, expressions, and values. In the directive descriptions Ada definitions are used. A number of definitions allow an address to be specified. In the directives below, Ada definitions are used. An address may be an Ada name, and Ada program statement or a machine address. Its syntax is:

address ::= name | name.integer | integer | @ address

The formats for directives are summarized below:

load object_description [parameters...]

Used to prepare a program object for execution. The parameters are specified using the CLI conventions.

goto [address] [, (*statements* | instructions) integer_expression]

Used to begin (or resume) execution. If the address is omitted, execution will begin at the program entry point (or resume where last interrupted). This directive may also be used to execute a specified number of statements or instructions.

help {action}

Used to obtain help information for the various debugger commands.

use package_name {,package_name}

This directive makes a name declared in a package referenceable without the package qualifier.

expression ?

The directive computes and displays the value of the expression or variable. Values displayed will be formatted according to the expression type. When displaying a variable, applicable indices or component names will be displayed.

variable'address ?

Used to print the address of a variable.

variable_name := expression

This directive is used to set a variable to the value of the expression.

trace [,(proc | flow | statement | assembly | walkback | off)]
[address]

Used to trace execution of the program. The options (which may be abbreviated by their first letter) are p for subprograms, f for flow paths (sequences of statements uninterrupted by flow control semantics), s for statements, a for assembly language level, w for subprogram walkback, and o for turning off indicated traces. The address is the limiting address of the traces.

dump [,(integer | real | boolean | binary | octal | hex | character |
enumeration | assembly | status)] address_range

Dumps the address range specified in the format requested: integer, real, Boolean, octal, hex, character, assembly instructions, hardware status.

(flow | time[,(proc | flow | report)] subprogram_name {,subprogram_name}

Causes path entrance frequency or timing information to be collected for the subprogram (or at the flow path level). The r option produces a summary report of the collected information.

insert (address; [directive {;directive}]) | ?

Effectively inserts the directives at the indicated program address. If the operand is ?, the active inserts are displayed.

handle (exception_name {, exception_name } [directive {;directive}]
) | ?

Effectively inserts the directives at the point where the exception is raised. If the operand is ?, the exceptions which the debugger is intercepting, and the exception which is being raised, if any, are displayed.

cancel implant_name {, implant_name }

Cancels a previous insert or handle.

record [, (read | write)] file , variable { , variable }

Used to write a variable to a file or to read a value into a variable. May be used to support environment simulation or post data reduction.

print text_object [line_range]

Used to list text objects such as Ada program source. By executing the Editor, the user may browse through his source program or enter his modification as debugging proceeds.

stop [,(status | walkback)] [expression]

Stops execution, displays execution status, active procedure call chain, and, on option, the value of expression.

quit

Terminates Debugger commands and returns to CLI.

In addition, Ada raise, abort, and delay statements may be used as directives.

17. TEXT EDITOR

The MDS text editor provides the ability to create and modify text files. Since the MDS will only be expected to support a limited number of users, it is reasonable to assume the availability of high-speed VDTs for editing. This is as opposed to the AIE, which needs the capability for editing by remote users over low-speed lines.

The editor design presented here is based largely on the CSC/SEA text editor. The primary difference is that the MDS editor was not constrained to provide support for low-speed or hard-copy terminals. There have also been some editorial changes to correct some typographical errors which appeared in the AIE version.

The design goals for this editor, aside from providing the basic editing facilities were:

1. The editor should be easy to use.
2. The commands must be natural, easy to learn and remember, but powerful and flexible.
3. The editor should provide screen-oriented editing facilities.
4. The editor should be as portable as is practical.
5. Support should be provided for minimal word processing and documentation generation.

There are a number of text processing systems available on microprocessors. In a number of cases the microprocessor-based text editors are easier to use than the editors which are hosted on larger systems. The text editor is a tool which is likely to be even more effective on a microprocessor based system than on a large host system, due to the fact that it is possible to assume the existence of high-speed VDTs for the MDS, while it is not always possible to do so for a large host.

For the purpose of the basic MDS we ruled out some of the more exotic forms of editors, such as syntax-directed editors [Morris81, Teitelbaum81] or editors which require special hardware such as touch panels or mice [Wadlow81]. There is no reason why these other types of editors could not be added to an MDS system; they just did not seem appropriate for the basic system.

This editor design has been influenced by a number of sources:

1. CSC/SEA AIE Editor- This was the largest single influence.

2. Intermetrics AIE Editor- This editor had an influence on the screen-oriented commands.

3. UCSD Pascal- This editor had an influence with respect to screen-oriented editing, environment specification and automatic indenting.

4. SOS- This editor had an influence on the line-range-oriented command.

5. Wordstar- This is a CP/M based word processing system. Its influence was in the area of screen-oriented editing.

6. CSTS Editor- This CSC editor has a somewhat unnatural syntax but permits a list of disjoint ranges, provides a means of referencing lines by means of fractional keys, and allows vertical windows in lines.

7. GCOS- This editor allows the ability to express a long, complex string with a simpler string range syntax.

8. TECO (Several Versions)- TECO provides basic editing facilities plus mechanism for combining commands in user-defined macros.

9. Jovial Users Group (JUG) AIE reviews- These comments pointed out a number of deficiencies in the original AIE document.

An editor is vital to program development, because it provides a means for creating source programs, and data and documentation files. The editor is, however, perhaps the most easily replaceable tool in the system since it is relatively simple compared to tools such as compilers and since its interfaces are simple - it must read and write standard text files. It is also one of the most controversial tools. (Witness the letters to the editor in [CACM 24:6, June 81] in response to [Ledgard80].)

The editor design presented here is by no means a minimal text editor. However, any system which can support an Ada compiler should have no trouble supporting the editor. There are a number of features in the editor design. Those that are most noteworthy and their reasons for being are discussed here.

One important feature of the MDS editor (or more properly, of the MDS system) is that lines in text files may be referred to by line numbers. This provides a static means of referencing lines in the editor, compiler and debugger. It also provides a convenient way to describe minor source changes for release to different systems when the changes are not large enough in scope to warrant an entire new system.

The editor is primarily screen-oriented, but contains non-screen-oriented commands to facilitate its use through command files, and to make it easier to make wholesale, systematic changes to a text object. The normal mode of operation is insert mode, where printable characters are entered in the file when they are typed, but where certain commands such as cursor movement and character, word and line deletion are available. Exchange mode allows characters to be typed in, replacing the text at the current cursor position. There is also a command mode which allows non-screen oriented commands to be entered.

The syntax of editor commands is a rather controversial subject. Options include menu selection, clear but verbose commands, concise but cryptic commands, or a combination of these approaches. There have been a number of studies of the effect of different styles of editors on productivity [Ledgard80, Roberts79, Robertson81, Schneiderman80, Card80]. Since it has been shown that the speed at which editing tasks are performed by experienced users is related to the number of keystrokes required [Card80], we have attempted to design a syntax which requires a minimum of keystrokes for common operations. For less common operations such as copies, reads, and writes, we have added some noise characters for increased clarity.

One of the more innovative features of the editor is the interface with the Ada compiler. The compiler generates error messages in a form suitable for processing by the editor. When the compilation error flag is set using the environment command, error messages are displayed along with the text. The editor also contains a facility for advancing the cursor to the next (or previous) line on which an error was detected.

Another editor feature is the buffer. Buffers may hold small amounts of text or entire text objects. They allow multiple files to be edited simultaneously, facilitating the movement of text from one database object to another.

Control constructs are provided so that editor commands may be executed repetitively or conditionally.

A macro facility is provided so that commonly used operation sequences may be performed with a minimum of keystrokes. An escape sequence provides access to the parameters which were passed to the macro. Macros are defined using the normal buffer manipulation commands to enter the text of the macro into a buffer.

The editor allows commands to be executed from an initialization file when the editor is first invoked. This allows a standard environment to be set up and provides a painless method for making standard macros available. These macros may be defined directly by commands in the initialization file, or the

initialization file may cause them to be read in from the database.

The editor is intended to be used primarily in screen mode; changes made to the work buffer are reflected instantaneously on the screen. The editor has several sub-modes of screen mode. In insert mode printable characters are entered into the work buffer at the current cursor position. In exchange mode printable characters replace the characters at the current cursor position. Mark mode allows a portion of the work buffer to be marked so that it can be treated as a unit by various command mode commands. In any of these modes it is possible to use the commands for moving the cursor, deleting text, changing the case of characters or swapping character positions, as well as searching forward or backward for strings.

There are three types of editing which are accomodated in the editor: program development, document preparation, and picture generation. These have many characteristics in common, but also require some special features.

Line numbers are useful primarily for program development. Screen-oriented commands can be made to act on lines of text. There is also the ability to act on a statement (text up to the next semicolon).

Operations applied to words and sentences are useful primarily for document preparation. Case changing operations fall into this category also. A novel feature of this editor is the ability to refer to a rectangular piece of text in a rangelist. This allows tables to be more easily manipulated. For example, it is possible to change a table from two columns wide to a single column simply by moving the text in one column in a single instruction.

It is sometimes desirable to be able to create pictures with an editor, for instance, in producing a document. This editor has a special feature to facilitate this. It is possible to define a current direction for the cursor to move when a key is typed. To produce a box, for example, it would be possible to set the direction to up or down to create the sides of the box simply by typing vertical lines.

Since the screen-oriented commands are difficult to illustrate on paper, the examples shown below are of the non-screen-oriented commands.

F Find

Find a string

FABC/	Find the first occurence of the string "ABC searching from the current position to the end of the work buffer.
-FALPHA/	Find the string "ALPHA" searching backward from the current position to beginning of the work buffer.
FBETA/300..500	Find the first "BETA" in lines 300 thru 500 inclusive.
F/	Find the most recent search string searching from the current position to the end of the work buffer.

D Delete

Delete text

D 100	Delete line 100
D100..200	Delete lines 100 thru 200 inclusively
D320#11..15	Delete characters eleven thru fifteen inclusively on line 320.
D61,37,43	Delete lines 61, 37 and 43.
3D	Delete three lines starting at the current line.
D*	Delete the distinguished Zone.

S Substitute

Substitute for string-1 string-2

SABC/DEFG/200	In line 200 substitute for all occurences of the string "ABC", the string "DEFG".
SALPHA/BETA/	Substitute for the next occurence of the string "ALPHA", the string "BETA".
SXYZ//	Delete the next occurence of "XYZ".
SABC/DEF/\>GHI/..\<JKL/	Substitute "DEF" for each "ABC" between the next occurence of "GHI" and "JKL".

C Copy

Copy text to a position, the source text is not deleted.

C30..60	Copy lines 30 thru 60 inclusive to the current position.
C200..250,100 TO 710	Copy lines 200 thru 250 and 100 inclusive to the position immediately after line 710. If line 710 does not exist the first line copied is given line number 710.
C*	Copy the distinguished zone to the current position.
C\BP	Copy the text described by the range contained in buffer P to the current position.

M Move

Move text to a position, delete the source text.

M55	Move line 55 to the current position.

M10,333,111 Move lines 10,333 and 111 to the current position.

M* Move the distinguished zone to the current position.

For the following example assume that the work buffer contains:

```
1000 1 4
1100 2 5
1200 3 6
```

Then the command:

```
M1000..1200#3 TO 1300
```

would give the result:

```
1000 1
1100 2
1200 3
1300 4
1400 5
1500 6
```

I Insert

Insert text at the current cursor position.

In this example text typed by the editor is underlined, <CR> represents carriage return, and < represents backspace.

```
500;I<CR>
500  if A = B
600      C := D;
700      E := F;
800      <<<end if;/
```

Note the auto-indenting in lines 700 and 800, and how in line 800, the user backspaced to get rid of the supplied blanks.

N Number

Number a range of lines.

N	re-Number the whole work buffer with current initial number and current increment.
N100..200,110,5	re-Number lines 100 thru 200 inclusive, start numbering at 110 in increments of 5.

R Read

Read (a portion of) an Ada Object into the work buffer, only text objects may be read.

RDEV_TAB/	Read the contents of text object DEV_TAB into the current position.
RTASK_TT/0..99	Read the text between lines 0 and 99 inclusive from text object TASK_IT into the current position. Neither line 0 nor line 99 need exist.
R\B1	Read the file or buffer named by the string contained in buffer 1 into the current position. If the buffer 1 contained the seven character string "DEV_TAB", this command would be the same as the first example above.

W Write

Write (a portion of) the current work buffer to a text object. If the text object is not empty, the output will be appended to the object.

WTPT_A/	Write the work buffer to TPT_A.
WTQAA/5,100..200,13	Write lines 5, 100 thru 200 inclusive and line 13 to TQAA.

B Buffer

Switch the current work buffer to a specified buffer.

BA	Switch to buffer A. The text in buffer A acts as the current work buffer.
B<CR>	Switch back to the default work buffer.

Position Cursor

Position the cursor to a specified location.

100	Cursor set to line 100.

P Print Text

Print specified text

P200	Print line 200.
P400..415	Print lines 400 thru 415.

A Assignment

Assign a value to a buffer

AI\BI+1	Increment the value contained in buffer I.

E Environment

Set/list environmental parameter

```
ESfill/1/              Turn line filling on.
```

18. OTHER TOOLS

An actual MDS installation is likely to have numerous tools beyond the ones described in this report. The tools discussed in this chapter are those which are necessary to implement the MDS, are of such general use so as to be worthy of inclusion in the basic MDS toolkit, or are tools which are closely related to the Ada Programming Support Environment.

18.1 MDS support tools

LALRGEN-parser generator

LALRGEN;

Since the compiler specification calls for a LALR parser, it is necessary to have a LALR parser generator for compiler maintenance. This tool would only be required on systems on which the compiler was maintained, but could certainly be used for other parsing applications even on systems where compiler maintenance was not performed.

LALRGEN's input is a text object containing the grammar for which tables are to be constructed. The output is a text object containing tables suitable for inclusion into the skeleton parser.

18.2 Additional MDS tools

The tools described in this section are those which are not quite so basic as the compiler, linker or editor, but which are important enough enough to be included in the basic MDS toolkit.

18.2.1 COM_LIST - generates compilation listings

```
COM_LIST (includes : boolean := true;
          notes: boolean := true;
          errors : boolean := false);
```

COM_LIST generates compilation listings from a source file (plus include files) and the error file generated by the compiler. Several listing options are available.

INCLUDES - if this flag is true, the text from objects named in include pragmas is listed.
NOTES - if this flag is true, notes are printed; otherwise, only

messages of level warning or higher appear on the listing.

ERRORS - if this flag is true, only lines containing errors are printed.

18.2.2 XREF - cross reference generator

XREF generates a cross reference listing from the debugging tables which are stored in the program library.

18.2.3 COMPARE - object comparison

COMPARE (file1: filename; file2: filename);

COMPARE generates a listing of the differences between two text objects on the standard output file. There is no standard input file.

18.2.4 IL_LIST - IL prettyprinter

IL_LIST (structure: boolean := false);

The IL lister converts the IL to a readable format: standard Diana format or a readable representation of the actual IL format. The library IL file is the standard input; the readable IL is written to standard output.

STRUCTURE - if this flag is true, the actual structure of the IL is printed; otherwise, it is printed as described in [Diana81].

18.2.5 FORMAT - text formatter

Text formatters are useful for preparing documentation. There is no doubt that microcomputers can host such tools. This document was prepared using a formatter which is an extension of the one described in [Kernighan76]. That formatter was, in turn, loosely based on Runoff [Saltzer65]. These formatters require the use of a text editor to create a file containing text and formatting commands. That file is run through the formatter to produce the document. This style typically uses a special character at the start of a line which contains commands. A variation on this is described in [Barach80]. Files for submission to this formatter more closely resemble the final document than do the files required by the formatter which was used for this report.

Another style of formatter is that which is commonly found on word processing systems where the text editor and formatter are a single program. Commonly,

the text is always displayed in formatted form; the user never sees the commands themselves, only their effects. Two examples of such formatters which are currently available on microcomputers are Wordstar [Micropro80] and Magic Wand.

Any of these styles of text formatters could be used with the MDS.

19. Performance Estimates

In order to predict the performance of a compiler on a new system, it is helpful to have the result of baseline compilations, and a measure of the relative speeds of the machine on which the baseline runs were made, and the machine whose performance is to be predicted. As was mentioned previously, compilation statistics have been obtained for J73/I and J73 compilers on various models of the IBM 370 and DEC-10. In addition, figures were obtained for several Pascal compilers running on a Z80-based microcomputer.

In order to determine the relative speeds of the large hosts and a number of microprocessors, "compiler Gibson mix" figures [Bloom74] were computed for the various machines. There are various instruction mixes which have been used to compute relative machine speeds [Bell78]. Commonly used measures of machine speeds are kilo operations per second (KOPS), million instructions per second (MIPS) and Whetstones [Curnow76]. The difficulty with such measures is that the relative power of different machines may vary depending upon the constituents of the job mix. Additionally, if the instruction mix which is used to determine the performance index does not accurately reflect the use to which the machine will be put, the performance index can be misleading. In an attempt to produce an accurate indication of how suitable the current microprocessors are for hosting compilers, several different methods of computing relative machine merits were used.

19.1 Compiler "Gibson Mix" from [Bloom74]

The "compiler Gibson mix" used here is based on a procedure described in [Bloom74] allowing for the differences in hosts when evaluating compiler performance. Our aim was to make use of the inverse relationship: if the compiler is held constant, what will its performance be on various host computers? The original "compiler Gibson mix" procedure involved hand-generating code sequences for the various host computers, calculating a weighted average for the various types of statements, and computing the averages for those types of statements (ifs and fors) whose averages depended on the overall average. The weights used for the various statements depended on the frequency with which those statements appeared in several actual compilers. Thus, a statement which was of a commonly occurring form (such as X = 0) would have a higher weight than a rarer type. Given that the weights accurately reflect the composition of the actual compiler, the relative performance index would be more accurate than that obtained using figures such as clock speed or add times for comparisons.

There are a number of shortcomings in the original "compiler Gibson mix". An attempt was made to correct the most significant weaknesses for the purposes of this study. Those deficiencies noted are:

1. For some machines and repetition counts for loop executions, numbers which represent contraditions are obtained.

2. The method which was used to measure the frequency of execution of the various statement types, ignored loops which did not include subroutine calls.

3. Subscripting and based data are ignored.

4. No allowance was made for the relative scope in which operands were defined.

5. The data types of operands were largely ignored. There were no character tests and no part-word extracts or deposits.

6. No allowance was made for input/output parameters.

7. No allowance was made for the effects of the architecture on a global basis.

The original compiler Gibson mix obtained figures for dynamic statement execution based on the static occurrence of the various statement types. The static frequency of occurrences of the different types of statements following thens, elses and dos was noted and used to produce equations which purported to reflect the dynamic execution frequency of the statements. As an example, the raw equation for the average execution time for a statement is based on the product of the number of times the average for loop is executed and the average time per statement. Thus, the average times for for statements, for if statements and for all statements depend on each other.

The authors of [Bloom74] suggest obtaining these three values by solving simultaneous equations. When this was done, however, negative values for the times were obtained. Obviously, this indicates that the model does not give good results, since negative average times are meaningless. The apparent reason for this is that divergent infinite series are being summed. This is due to the fact that if the repetition count for for loops is sufficiently large, the time for inner loops is sufficiently great that, despite the fact that only one statement in 50 is a for loop, the time contributed to the average by a for increases as the nesting level increases.

The problem stems from the fact that static statistics were used as the basis for dynamic statistics. Our solution to this problem was to observe that in the limiting cases, the statements in loops are never executed ($r = 0$) or the statements in loops are executed so many times that any statements not in loops are executed with such a low relative frequency that hey are

insignificant. Thus, at the two extremes, the frequencies of executions for the various kinds of statements should approach the frequencies for statements outside loops and inside loops, respectively. The averages were calculated using both of these sets of weights. It was not expected that the two sets of averages should show any significant differences, and in fact, this turned out to be the case.

It should be noted that this means of determining dynamic frequency of execution is approximate, at best. A more satisfactory method would be to compile statistics dynamically from execution frequency histograms. The raw data for such computations were not available, so the static data were used as a compromise. Using dynamically acquired data would give a better picture of the actual frequency of execution, provided that the statistics were obtained using good benchmark data.

Subscripting and basing of data were ignored in the original compiler Gibson mix. Since the original statistics did not include a measure of the frequency of the occurrence of these operations, the performance indices were originally calculated assuming the absence of subscripting and a correction factor was added in later. The weight of this correction factor was based on the frequency of subscripting operations which were observed in J73 compilers as part of this study rather than those compilers which were analyzed in [Bloom74]. Note that the architecture has an effect beyond that of the time for a single indexing operation, since, for some machines which do not allow data to be based and indexed, an additional add instruction must be generated.

The mix did not take the scope of operands into account. This is probably not a serious deficiency, since other studies [Wirth71] have shown that the majority of references are to data that is either local or global.

The data types for operands were ignored for the most part. The mix deals exclusively with integers and booleans, and booleans appear only in if statements. Character data and floating point data do not appear in the mix. The absence of floating point operations is quite plausible, since compilers typically use floating point operations only for performing constant arithmetic. Therefore, both their static and dynamic frequencies of occurrences should be insignificant. The omission of character data is more questionable. Although the static frequency of occurrences of character operands is low (since character data is involved primarily with reading source and generating listings and these functions tend to be isolated), the dynamic occurrence frequency may be quite high in some cases [Earnest70].

Another problem with operand data types is that no figures were included for the occurrences of part word operands. These may occur quite often if a packed symbol table structure is used.

There was no allowance made for the occurrences of input and output parameters. All the parameters in the mix were input parameters. This should not be a very significant factor. Since no statistics were available, no correction was attempted.

Another deficiency, and one which would be very hard to remedy, is that there is no allowance for the effect of the different machine architectures on a global basis. There is no real advantage in the mix for a machine to have a number of registers. In order for the effects of such machine features to be measured, it would be necessary to have such information as the number of common subexpressions live at a given time. Other machine features, such as short branch instructions or skips, have an effect which can be approximated by making some reasonable assumptions based on empirical data [Peuto77].

There are a number of statistics which would be useful in more accurately characterizing JOVIAL usage. For a comprehensive list see [Greenspan80].

19.2 "Gibson Mix" Calculation

A number of assumptions were made in computing the performance indices for the various computers. These assumptions are given here:

1. Code must be recursive and reentrant. (This is a result of the Ada requirement that procedures be recursive and reentrant.)

2. Operands are local (or global) and the appropriate pointer is available without loading.

3. Local variables are few in number (< 256 address units). Short addressing forms are used, if available.

4. Conditional branches are relatively short (but not very short). Short relative branches are used but skips are only used in "canned" code sequences.

5. Short code sequences are better than fast code sequences, except that subroutine calls are avoided.

6. It must be possible for a production quality compiler to generate the code sequences.

7. Parameters are passed on the stack.

8. Register saving and restoring is ignored, except for environmental pointers.

The code sequences generated for the mix are both recursive and reentrant because Ada requires that procedures possess these attributes. This is not a requirement for all J73 procedures (although it may optionally be specified). It was thought that the Ada requirements are more important.

The effect of this particular requirement is relatively insignificant except with respect to the Z80 and the IBM 370. The Z80 is not well-suited to such operations, since not all instructions may be indexed. In particular, the lack of 16 bit indexed load instructions makes the performance of the Z80 worse, relative to the other machines, than if the code were not required to be reentrant. It should be noted that the 8080 would fare worse than the Z80 due to this requirement. The 370, on the other hand, allows many instructions to be both based and indexed; its performance is better, relative to the other machines, than if this requirement were dropped.

The assumption that the local and global scope pointers are available means that the pointers are in registers for machines which have them, and available implicitly for the P-machine, which has no registers. None of the architectures which were studied had trouble meeting this requirement, although the 8080 would have had some difficulty due to the fact that the HL register is the only index-like register on the machine, except for certain special cases. (The BC and DE pairs may be used to address memory to load or store the accumulator, but only HL may be used to hold the address for arithmetic operands.)

The assumption that most branches are short is a reasonable one [Peuto77]. The only short branch with a range smaller than 128 bytes in the architectures studied is the DEC-10 skip. It was thought that it was unreasonable to use skips, except for "canned" code sequences (e.g., where it was known that a branch instruction followed the skip). In certain other cases it would be possible to achieve optimization (e.g., IF condition THEN statement, where "statement" happens to require one word of code) but it was thought that the effect of this sort of optimization would be quite small, relative to the precision of the performance indices.

Where a tradeoff had to be made between code sequences, the shorter of the two was chosen, the rationale being that shorter sequences always save space but faster sequences only save time when they are executed [Wulf75]. The exception to this rule was that sequences for all of the statements (except for call statements themselves) were generated inline. This was done because the intention was to measure the suitability of the machines for hosting a compiler, rather than the machines' efficiency for subroutine calls. The only machine for which this had much effect was the Z80, since the other machines had sufficiently powerful instruction sets so that it would be natural to generate the code in-line.

It is assumed that the compiler would be self-compiling. While it would be possible to achieve some savings by coding the compiler in assembly language, the increase in development costs, particularly considering the size of Ada, would be prohibitive. Thus, it would not be meaningful to use code sequences which could not be generated by a production quality compiler. The only sequences which would not be generated by a compiler such as the SEA J73 compiler are the for loop sequences. These sequences dedicate a register for the loop variable, whereas the SEA J73 compiler does not perform such optimization, although it is well within the state of the art.

In order to simplify the computation of the mix, it was assumed that parameters would be passed on the stack, except for those machines (the 370 and the 1750A) which had no stack support. The call statements and prologues give the greatest freedom for varying code sequences, parameters in the stack, list parameters, or registers. A complete analysis of the best method for each machine would have taken too much time so the stack method for parameter passing was chosen as an expedient.

Procedure prologues and epilogues for general register machines normally include code to save and restore register contents. This saving and restoring was ignored for the purpose of computing the performance indices. One reason is that it was not really possible to take the registers into account with respect to common subexpression elimination. It would be unfair to penalize a register machine for having a place to save common subexpressions by including save and restore code without rewarding it when the subexpressions were used.

The method of computation of the performance indices was as follows:

1. Hand-generate code sequences for 65 types of statements for each machine architecture.

2. Tally the time and space costs for each statement. Timings for the P-machine were done experimentally. Space and the remainder of the times were calculated from the vendor's hardware manuals.

3. Weighted averages were computed for each class of statements (assignments, ifs, etc).

4. The overall performance indices were calculated, using weighted averages of the averages for different statements. Weights were taken from [Bloom74] and from Appendix I of this report.

Results obtained using the Bloom mix and the mix from Appendix I are almost identical, which is not surprising since the percentages of different statements are quite similar. Another study [Shimasaki80] presented statement mix results for various Pascal compilers. There the percentage of calls was

higher and the percentages of ifs and assignments were lower. A summary of results of studies of the characteristics of high-level language programs may be found in [Softech80].

The performance indices for the microprocessors were calculated in terms of clock cycles, since most of processors are available with several clock speeds. In addition, faster models of the same microprocessors may be introduced in the future.

Cycle counts may be converted to speeds by dividing by clock frequency. Thus, for an 8 MHz MC68000, the 43 cycles would represent 5.4 microseconds, but for an 4 MHz microprocessor, it would represent 10.8 microseconds.

19.3 Compiler Performance Estimates

The theoretical machine speeds in Appendix M and the benchmark execution times in Appendix J indicate that the more powerful microprocessors are comparable to some of the slower models of the DEC-10, at least with respect to CPU speed. KOPS figures obtained from various sources also indicate this to be true. The DIS is rated at 350 KOPS (4 MHz) or 525 KOPS (6 MHz) [General Dynamics80] and various models of the DEC-10 are rated at 166 KOPS (KA10), 497 KOPS (KI10), and 829 KOPS (KL10) [Lias80]. These figures do not, though, take the differences in word size into account. Thus, in projecting the performance of a microcomputer-hosted compiler, it is not sufficient to compute a ratio of machine "horsepower" to use as a multiplier. Benchmark results and mix figures do provide a place to start, however.

The 68000, on which the benchmarks were run, falls somewhere between the KA10 and KI10 with respect to CPU speed. Assuming that the effective speed of the 68000 was 4 MHz due to bus speed limitations, the 68000 is between these two DEC-10 models in theory, also. Compilation speeds ranged between 850 and 1250 lines/minute on the KA10 and 2000 to 2900 on the KI10.

It should be noted that compilation speeds for languages such as JOVIAL and Ada are less meaningful than figures for other languages due to language features, such as compools and packages, which can require great amounts of processing for few source lines. The estimates made here assume "normal" use of compools or packages, as well as reasonable formatting conventions (such as statements/line, and comments/statement).

The figures in Appendices A and K indicate that the bulk of the I/O performed while the J73 compiler is executing, is that required to read in the compiler itself. The compiler phases are in the neighborhood of 300K bytes, but the size of intermediate files (such as IL) is in the 64K range even for large modules. This indicates that a reasonable approximation of compilation speed

may be obtained by dividing the number of lines compiled by the sum of the CPU time and I/O time. Note that this applies only to a particular compilation strategy, one in which most of the I/O cannot be overlapped with computation. In addition, a virtual memory strategy for the symbol table could significantly increase the amount of I/O required. This would invalidate the assumption that most of the I/O was due to phase loading.

The KA10 speeds are the best place to start for calculating compilation speeds for a 68000 hosted compiler, since the raw machine speed for the KA10 is somewhat slower than that of the 68000. This allows for the fact that there are some differences in wordsize. (The 68000 allows most operations to be performed on 32 bit operands but lacks some 32 bit operations such as multiplication and division.)

Although there are twice as many phases in the proposed Ada compiler as in the SEA J73 compiler, the individual phases are not as complex. Although Ada requires more processing than J73 (primarily due to the more complicated visibility and overloading resolution rules), the compilation rates should be comparable, because the J73 compiler provided additional facilities which are in separate MDS tools. For the purposes of this analysis we will assume that a 1000 line module is being compiled. This would require about a minute of CPU time on a KA10, based on the figures in Appendix D.

The I/O required for the Ada compiler is greater than that for the J73 compiler for several reasons. First, the compiler has more phases, so the IL must be read and written more times. Second, the symbol table is not memory-resident. Third, the Ada IL (DIANA) contains more information than does the J73 IL. (This is partially, but not totally, due to the fact that Diana contains information comparable to that in the J73 symbol table, as well as the IL.) Statistics from the University of Karlsruhe indicate that their compiler generates 3-5 IL nodes/lines of source and that the average node contains 30-40 bytes. These figures are for the Aida IL, but should be a good approximation for Diana, since Diana is a descendant of Aida and the two are quite similar. For our thousand line module, the IL size would be in the vicinity of 90,000-200,000 bytes. The University of Karlsruhe Compiler distributed the attribute information in the terminal nodes, rather than having small nodes which pointed to a symbol table type structure. This was done to maximize locality (and minimize page faults) on a host with virtual memory. This is not advantageous for current microcomputer architectures, so the compiler design presented here does use a symbol table. This implies that the actual size of the IL should be somewhat smaller.

Equivalent estimates for the J73 IL and symbol table sizes for a thousand line module based on figures in Appendix K are 14,000 bytes (14 bytes/statements * 1000 lines) of IL and 16,000 bytes of symbol table for a total of 30,000 bytes. The figures in Appendix K give bytes of IL per statement (excluding

declarations). The calculation made here uses that figure multiplied by the total number of lines, since IL is generated for declarations as well as the normal procedural statements. These figures are exclusive of names (which won't appear in all versions of the IL) and serve as a sort of lower bound on the amount of IL to be generated, since Ada would be likely to require more IL than J73.

The IL (and AST) are read and written a total of 14 times (7 reads, 7 writes) by the proposed compiler. Assuming 100,000 bytes of IL, this represents 1,400,000 bytes read and written. In addition, there are 9 memory loads of roughly 100,000 bytes each for another 900,000 bytes read. Other files read include source (1000 lines at 40 characters/line of 40,000 characters), the code file, the relocatable, and debugging information written to the program library.

The total number of bytes read or written for the 1000 line module is roughly 3,000,000. Assuming 256 bytes per sector, this represents roughly 12,000 sectors read or written.

General Assumptions:

Size of module being compiled = 1000 lines
"Average" use of packages
CPU use = 1 minute
Total size of compiler phases = 1MB
Total size of data read/written (excluding phases) = 2MB
Compilation-time = CPU-time + (Disk-access-time + transfer-time)

Assuming the worst case for the number of seeks (each sector requiring access time), for a typical 8-inch Winchester disk with average access time around 80ms and a transfer rate of 500K bytes/second, transfer time would be 6 seconds and access time would be 960 seconds. This would give a compilation speed of under 60 lines/minute (wall clock). (In comparison, an IBM 3370 has an average access time of about 30ms and a transfer rate of around 1900K bytes/second [Mini-Micro Systems, February 1981, p. 111].)

Fortunately, more realistic assumptions yield more favorable results. First of all, disk accessing is not random. The compiler phases are read sequentially, largely eliminating the seek time for these sectors. The compiler's temporary files are read sequentially, for the most part, so it is not necessary that there be a seek for each sector read. Assuming that there are 32 sectors/track, the seek time should only come into play between 10 and 15 times per phase load (since a phase should fit on 10-15 tracks). This would involve an average seek for the initial read (80ms) plus about 450 ms total seek time for the remaining reads. (This assumes that subsequent seeks for a phase only require the head to move one track.) Since there are 10

memory loads, the total seek time for phase loading would be around 5.3 seconds.

Other compiler I/O is not as simple to analyze, since the pattern of accesses cannot be determined a priori with the same degree of certainty. Reading or writing four sectors at a time would reduce the number of seeks to 2000 (for the 8000 remaining sectors). The actual number of seeks required would depend on the pattern of references to the input and output files. A seek for every four sectors would give a seek time of 160 seconds for the temporary files and a compilation speed of slightly under 260 lines/minute (wall clock). Consecutive references to the same file would decrease the seek time and increase the compilation speed.

Assumptions:
- Sector size = 256 bytes
- Sectors/track = 32
- Average access time = 80 ms
- Track to track time = 20 ms
- Transfer rate = 500KB/sec
- Seek for each sector

Results:
- Transfer time = 6 seconds
- Total I/O access time = 960 sec
- Compilation speed = < 60 lines/minute (wall clock)

Assumptions:
- Phases read sequentially (one seek per phase)
- Data read in blocks of 4 sectors

Results:
- Transfer time = 6 seconds
- Total I/O access time = 165 sec
- Compilation speed = < 260 lines/minute (wall clock)

The corresponding calculation for a relatively fast 8" floppy disk drive, such as the Shugart Associates SA851 gives the following results:

Assumptions (8-inch floppy disk):
- Sector size = 256 bytes
- Average access time = 210 ms
- Transfer rate = 50KB/sec

Results:
- Transfer time = 60 seconds
- Total I/O access time = 2520 sec

Compilation speed = < 40 lines/minute (wall clock)

Assumptions:
Phases read sequentially (one seek per phase)
Data read in blocks of 4 sectors

Results:
Transfer time = 60 seconds
Total I/O time = 440 sec
Compilation speed = < 110 lines/minute (wall clock)

The I/O characteristics of the host system have a significant effect on compilation speeds. It is unlikely that a system with floppy disks can provide sufficient throughput for an acceptable compilation speed to be achieved. A system with Winchester disks, however, could provide acceptable performance.

The estimates for the system with the Winchester disk are rather conservative since disks with faster access times and higher transfer rates are available. In addition, a 68000 running at full speed should actually be faster than the KA10 for compilations. Also, any overlapping of CPU use and I/O would increase the speed.

20. CONCLUSIONS

Current microcomputer systems are sufficiently powerful to be able to host systems for serious software development. Compilation speeds in the neighborhood of 250 lines/minute (wall clock) should be attainable for compilers which generate optimized code. Speeds of over 550 lines per minute have already been obtained for a microcomputer-hosted Ada compiler generating unoptimized code.

There are several presently-available microprocessors around which it is possible to build such a development system. The Motorola MC68000 is most suited for such a system due to its large address space and 32 bit operations. The Zilog Z8000 and Intel 8086 could also be used as the basis for such systems. The Intel 432, which should be available in the near future, will easily support an MDS. The National Semiconductor 16032 will also be a suitable host when it reaches the market.

It should be noted that not all microcomputer systems are suitable for hosting an Ada compiler and development system. In particular, many of the hobbyist type microcomputers suffer from both main memory and peripheral storage limitations.

Even the high-end microprocessors are quite inexpensive in relation to mainframe computers. With respect to cost effectiveness, the biggest competition for microcomputers comes from the so-called "superminis" such as the VAX and its competitors.

We have presented here and in companion volumes a design for a development system containing such tools as a command language interpreter, an Ada compiler, a linker, a debugger and a text editor, as well as the operating system itself and the database system. This system is implementable on current microcomputer hardware.

The fact that microcomputers are so inexpensive makes it possible to think about the allocation of resources in different ways compared to the past. Since computer power is now relatively cheap compared with a programmer's time, it becomes advantageous to provide hardware, above and beyond the minimum necessary, in order to save on labor costs. The development system described here straddles the gap between conventional timesharing and a network.

In the future hardware costs are likely to continue to decrease, relative to software costs, and there is likely to be a trend toward providing one (or more) CPUs per user. Improvements in hardware will make a microcomputer-based development system even more attractive in the future than it is today.

20.1 Future Work

The most important work which needs to be done is the implementation of a microprocessor-based Ada development system.

There are a number of other issues which need to be addressed. One of the most important is debugging, particularly for optimized code on a target which has limited capabilities. None of the AIE designs fully addressed these problems, and they have not been addressed here in as much detail as we would have liked. It seems that it should be possible for the optimizer to provide enough allocation information to the debugger, so that the debugger can know where a given variable resides during the execution of a particular portion of the test program. However, to our knowledge, no one has actually attempted to prove rigorously that this can be done.

Debugging on a target which has very limited capabilities is a particular problem for embedded computer system development. There are a number of issues here, such as how the host and the target communicate, and how the debugger manages to control the test program without being intrusive.

Program development environments deserve attention, both with regard to Ada-specific environments and environments in general. Considerable work is being done in this field. The AIE and ALS efforts should provide insight into what is needed for successful program development in Ada.

The Ada program library is an important part of the environment which needs to be considered in greater detail. There are conflicting goals with respect to ease of use and generality. A library that is too automatic may not be flexible enough; one that is very powerful may be too complicated. We have attempted to present a scheme that is a good compromise, but only experience will prove or disprove this conclusion.

21. References

Ada Implementor's Group, Ada Implementor's Newsletter.

Adatec, Ada Letters, formerly Ada Implementor's Newsletter.

Aho, A. and Johnson, S., "LR Parsing", ACM Computing Surveys 6:2, June 1974, pp.99-124.

Aho, A. and Ullman, J., Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1977, 604p.

Aho, A. and Ullman, J., The Theory of Parsing, Translation, and Compiling, Prentice Hall, Englewood Cliffs, N.J., 1972.

Albrecht, P., et al, "Source-to-Source Translation: Ada to Pascal and Pascal to Ada, SIGPLAN 15:11, November 1950, pp.183-193.

Allison, D., "A Design Philosophy for Microcomputer Architectures", Computer, Feb. 1977, pp. 35-43.

Backus, J., "The History of FORTRAN I, II and III", SIGPLAN 13:8, August 1978, pp. 165-180.

Barach, D. and Fran, G., "NNP: An Easy to Implement Preprocessor for Text Formatting", Software - Practice and Experience 10:5, May 1980, pp. 335-346.

Bell, J., "Threaded Code", CACM, June 1973, pp. 370-371.

Bell, Mudge and McNamara, Computer Engineering: A DEC View of Hardware Design, Digital Press, Bedford, Mass., 1978, 585p.

Bloom, B. et al, Criteria for Evaluating the Performance of Compilers, RADC Technical Report RADC-TR-74-259.

Boom, H., and De Jong, E., "A Critical Comparison of Several Programming Language Implementations", Software - Practice and Experience 10:6, June 1980, pp. 435-473.

Bourne, S., "The UNIX Shell", Bell System Technical Journal 57:6, part 2, pp. 1971-1990.

Boyer, R. and Moore, J., "A Fast String Searching Algorithm", CACM 20:10, Oct. 1977, pp. 762-772.

Brender, R., "The Case Against Ada as a Command Language", SIGPLAN 15:10, Oct. 1980, pp. 27-34.

Brender, R., and Nassi, J., "What is Ada?", Computer 14:6, June 1981, pp.17-25.

Brosgol, B. et al, TCOL-Ada: Revised Report on an Intermediate Representation for the Preliminary Ada Language, Carnegie-Mellon University, CMU-CS-80-105, 149 p.

Buxton, J., "An Informal Bibliography on Programming Support Environments", SIGPLAN 15:12, Dec. 1980, pp. 17-30.

Callahan, J., et al, "Bringing Virtual Memory to Microsystems", Electronics, June 30, 1981, pp. 119-122.

Card, S., Moran, J., and Newell, A., "The Keystroke-Level Model for User Performance Time with Interactive Systems", CACM 23:7, July 1980, pp. 396-410.

Carlson, W., "Ada: A Promising Beginning," Computer 14:6, June 1981, pp. 13-16.

Carter, J., "A Case Study of a New Code Generation Technique for Compilers", CACM 20:12, Dec. 1977, pp. 914-920.

Champine, G., "Current Trends in Database Systems", in Selected Reprints in Software, IEEE, 1980, pp. 196-210.

Cherlin, E., "The UNIX Operating System: Portability a Plus", Mini-Micro Systems 14:4, April 1981, pp. 153-159.

Clark, W., "From Electron Mobility to Logical Structures: A View of Integrated Circuits", ACM Computing Surveys 12:3, September 1980, pp.325-356.

Cocke, J. and Schwartz, J., Programming Languages and Their Compilers, Courant Institute, 1970.

Cole, S., "Ada Syntax Cross Reference", SIGPLAN 16:3, March 1981, pp. 18-47.

Comer, D., "The Ubiquitous B-Tree", ACM Computing Surveys 11:2, June 1979, pp. 121-138.

Computer, Ada Issue, June 1981.

Computer, Programming Environments Issue, April 1981.

Computer Automation, Scout Naked Mini 4/04 Handbook, January 1980.

Cragon, H., "The Elements of Single-Chip Microcomputer Architecture," Computer 13:10, Oct. 1980, pp. 27-41.

CSC/SEA, Ada Integrated Environments: Computer Program Development Specification, 1981.

CSC/SEA, Ada Integrated Environment: Interim Technical Report, 1981.

CSC/SEA, Ada Integrated Environment: System Specification, 1981.

CSC/SEA, Design Evaluation Report for the Ada Integrated Environment, 1981.

Crespi-Reghizzi, S., Corti, P., and Dapra, A., "A Survey of Microprocessor Languages," in Selected Reprints in Software, IEEE, pp. 47-65.

Dausmann, M. et al, An Informal Introduction to AIDA, University of Karlsruhe, AIDA-12.

De Remer, F. and Pennelo, J., "Efficient Computation of LALR (1) Look-ahead Sets", SIGPLAN 14:8, August 1979, pp.176-187.

Diana Reference Manual, Jan 20, 1981.

Digital, Microcomputer Interfaces Handbook, 1980, 736p.

Digital, Microcomputer Processor Handbook, 1980, 602p.

Digital Research, PL/I-80 Reference Manual.

Ditzel, D., "Program Measurements on a High-Level Language Computer", Computer 13:8, August 1980, pp. 62-78.

Dolotta, J., Wright, R., and Mashey, J., "The Programmers Workbench", Bell System Technical Journal 57:6, part 2, July-August 1978, pp. 2177-2200.

Donegan, M., "The Design of a Space Efficient Compiler", in Proceedings of the First SIGMINI Symposium on Small Systems, New York, August 2-3, 1978, SIGMINI 4:4, August 1978.

Earnest, C., in Cocke & Schwartz, Programming Languages and their Compilers, Courant Institute, 1970.

EDN, uP/uC Chip Directory, November 5, 1980.

Edwards, J., Tailored Jovial J73 for Small-Scale Development Projects in Avionics.

Electronics, April 17, 1980, 650p.

Elger, P., Some Observations Concerning Existing Software Environments, 39p.

Elshoff, J., "An Analysis of Some Commercial PL/I Programs", IEEE Transactions on Software Engineering, SE 2(2), 1976, pp. 113-120.

Embley, D., and Nagy, G., "Behavioral Aspects of Text Editors", Computing Surveys 13:1, March 1981, pp. 33-70.

Estell, R., "Benchmarks and Watermarks", Performance Evaluation Review, Fall 1980, pp. 39-44.

Forsyth, C., and Howard, R., "Compilation and Pascal on the New Microprocessors", BYTE, August 1978.

Franta, W., "JOYCE: A Next Generation Personal Computer", in Proceedings of the Third Symposium on Small Systems, SIGSMALL, Sept. 1980, pp. 108-113.

Fisher, C., and LeBlanc, R., "Efficient Implementation and Optimization of Run-Time Checking in Pascal", in Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12:3, March 1977, pp. 19-24.

Fisher, D., Design Issues for Ada Program Support Environments: A Catalogue of Issues, Science Applications, Inc. report SAI-81-289-WA, 21p.

Flynn, M., "Directions and Issues in Architecture and Language", Computer 13:10, Octs 1980, pp. 5-26.

Ganzinger, H. and Ripken, K., "Operator Identification in Ada: Formal Specification, Complexity, and Concrete Implementation", SIGPLAN 15:2, Feb. 1980, pp. 30-42.

General Dynamics, DIS: Digital Integrating Subsystem, 5 November 1980, 63p.

Geschke, C., Morris, J. and Satterthwaite, "Early Experiences with Mesa", CACM 20:8, Aug. 1977, pp. 540-552.

Gilbreath, J. "A High-Level Language Benchmark", BYTE 6:9, Sept. 1981, pp.180-198.

Glass, R., "Real-Time: The 'The Lost World' of Software Debugging and Testing", CACM 23:5, May 1980, pp. 264-271.

Goodenough, J., "The Ada Compiler Validation Capability", Computer 14:6, June 1981, pp.57-64.

Goodenough, J., "Ada (July 1980) Syntax Cross Reference Listing", SIGPLAN 15:10, Oct. 1980, pp. 48-56.

Graham, S., "Table-Driven Code Generation", Computer 13:8, August 1980, pp. 25-37.

Grappel, R., and Hemenway, J., "Benchmarks for 16-bit Processors", EDN, April 1, 1981, pp. 179-265.

Grappel, R., and Hemenway, J., "Compare the Newest 16-bit uPs to Evaluate Their Potential", EDN 25:16, September 5, 1980, pp. 197-201.

Greenspan, L., and Helimantel, M., Design Issues for the Advanced Avionic Computer Architecture, Sanders Assoc., 1980.

Gutz, S., Wasserman, A., and Spier, M., "Personal Development Systems for the Professional Programmer", Computer 14:4, April 1981, pp. 45-53.

Hall, D., Scherrer, D., and Sventek, J., "A Virtual Operating System", CACM 23:9, Sept. 1980, pp. 495-502.

Heckel, P., "A Technique for Isolating Differences Between Files", CACM 21:4, April 1978, pp. 264-268.

Hemenway, J., and Grappel, R., "Intel's iAPX 'Micromainframe'", Mini-Micro Systems 14:5, May 1981, pp. 73-89.

Heyliger, G. et al., Recommendations for a Retargetable Compiler, RADC technical report RADC-TR-79-351, 1980, 149p.

Hillsberg, B., "Generic Terminal Support", Operating Systems Review, 15:2, April 1981, pp.10-15.

Hoare, C., "The Emperor's Old Clothes", CACM 24:2, Feb. 1981, pp. 75-83.

InfoWorld, 2:15, Sept. 1, 1980.

Intel, Introduction to the iAPX 432 Architecture.

Intel, iAPX 432 General Data Processor Architecture Reference Manual, 1981.

Intel, The 8086 Family User's Manual, 1979.

Interface Meeting on Programming Systems in the Small Processor Environment, SIGPLAN 11:4, April 1, 1976, 164p.

Intermetrics/MCA, Ada Integrated Environment: Computer Program Development Specification, 1981.

Intermetrics/MCA, Ada Integrated Environment: Interim Techincal Report, 1981.

Intermetrics/MCA: System Specification, 1981.

Intersystems, Intersystems Pascal/Z A High Level Programming Language, Version 3.0, 1980.

Inui, N., Kikuchi, H. and Sakai, T., "16-Bit C-MOS Processor Packs in Hardware for Business Computers", Electronics 54:12, June 16, 1981, pp. 182-186.

Ivie, E., "The Programmer's Workbench - A Machine for Software Development", CACM 20:10, Oct. 1977, pp. 746-753.

Jensen and Wirth, Pascal User Manual and Report, Springer-Verlag, New York 1974, 167p.

Johnson, R., "Major Firms Join Unix Parade", Electronics 54:7, April 7, 1981, pp.108-112.

Johnson, R., "Microsystems Exploit Mainframe Methods, Electronics, August 11, 1981, pp. 119-127.

Johnson, S., "Language Development Tools in the Unix System", Computer 13:8, pp. 16-24.

JRT Systems, JRT Pascal User's Guide, San Francisco, 1980.

JUG, USAF-JUG Ada Integrated Environment Contractor Evaluation, 1981.

Kane, G., 68000 Microprocessor Handbook, Osborne/McGraw-Hill, Berkeley, Ca., 1981, 113p.

Kane, J. and Osborne, A. An Introduction to Microcomputers, Volume 3, Some Real Support Devices, Osborne & Associates, Berkeley, 1978.

Kennedy, K., "Node Listings Applied to Data Flow Analysis", 2nd POPL, 1975, pp. 10-21.

Kernighan, B., and Mashey, J., "The UNIX Programming Environment", Computer 14:4, April 1981, pp. 12-24.

Kildall, G., "A Unified Approach to Global Program Optimization", POPL, 1973, p.194-206.

Klingman, Edwin, Microprocessor Systems Designs, Prentice-Hall, Englewood Cliffs, N.J., 1977.

Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley: Reading, Mass., 1973, 723p.

Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software - Practice and Experience, 1971, pp.105-133.

Lamb, D., Construction of a Peephole Optimizer, Carnegie-Mellon University Report CMU-CS-80-141, Aug. 1980, 11p.

Lamb, D., "Construction of a Peephole Optimizer", Software - Practice and Experience 11:6, June 1981, pp. 639-647.

Lamb, D., et al, The Charrette Ada Compiler, Carnegie-Mellon University Report CMU-CS-80-148, Oct. 1980, 123p.

Lampson, B., and Sproull. R., "An Open Operating System for a Single-User Machine", Proceedings of the Seventh Symposium on Operating System Principles, pp. 98-105.

Ledgard, H. et al, "The Natural Language of Interactive Systems", CACM 23:10, Oct. 1980, pp. 556-563.

Lehman, M., Pascal/MT Release 3.0 User's Guide, MT Microsystems Cardiff-by-the-Sea, Ca, 1980, 93p.

Leverett, B. et al, "An Overview of the Production-Quality Compiler Compiler Project", Computer 13:8, August 1980, pp. 38-50.

Lias, E., "Tracking the Elusive KOPS", Datamation 26:11, Nov. 1980, pp. 99-105.

Loveman, D., "Program Improvement by Source to Source Tranformation", 3rd POPL, 1976, p.140-152.

Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures", CACM 20:3, March 1977, pp. 143-152.

Lycklama, H., "UNIX on a Microprocessor", Bell Systems Technical Journal 57:6, part 2, July-August 1978, pp. 2087-2102.

McCauley, E., Doka, E. and Baladi, N., "Challenging the Minis", Mini-Micro Systems 14:9, September 1981, pp. 135-140.

Mead, C., and Conway, L., Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980, 396p.

Metcalfe, R. and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM 19:7, July 1976, pp. 395-404.

Micropro International Corp., Word Star User's Guide, San Rafael, Ca., 1980.

Military Standard JOVIAL (J73), MIL-STD-1589B (USAF), 06 June 1980, 168p.

Miller, L. and Thomas, J., "Behavioral Issues in the Use of Interactive Systems", International Journal of Man-Machine Studies 9, 1977, pp. 509-536.

Mooney, C., in [JUG81].

Moran, T., "Guest Editor's Introduction: An Applied Psychology of the User", Computing Surveys 13:1, March 1981, pp. 1-12.

Morris, J. and Schwartz, M., "The Design of a Language-Directed Editor for Block-Structured Languages", SIGPLAN 16:6, June 1981, pp. 28-33.

Morse, S., et al, "Intel Microprocessors - 8008 to 8086", Computer 13:10, Oct.1980, pp. 42-62.

Motorola, MC68000 16-Bit Microprocessor User's Manual, 1980.

Nelson, V., and Nagle, H., "Digital Filtering Performance Comparison of 16-Bit Microcomputers", IEEE Micro 1:1, February 1981.

Noyce, R., and Hoff, M., "A History of Microprocessor Development at Intel", IEEE Micro 1:1, Feb. 1981.

Ogdin, C., Microcomputer Management and Programming, Prentice-Hall, Englewood Cliffs, 1980, 348p.

Osborne, A., An Introduction to Microcomputers: Volume 1 Basic Concepts, Osborne/McGraw-Hill, Berkley, Ca., 1980.

Osborne and Kane, An Introduction to Microcomputers, Vol. 2, Osborne & Assoc.,

Berkeley, CA, 1979.

Osterweil, L., "Software Environment Research: Directions for the Next Five Years", Computer 14:4, April 1981, pp. 35-44.

Persch, G., et al, AIDA Reference Manual, University of Karhruhe, AIDA:14.

Persch, G., et al, "An LALR (1) Grammar for (Revised) Ada", SIGPLAN 16:3, March 1981, pp. 85-98.

Peuto, B., and Shustek, "Current issues in the Architecture of Microprocessors", Computer Feb. 1977, pp. 20-25.

Phase One Systems, OASIS System Reference Manual, Oakland, 150p.

Poe, E. and Goodwin, J., The S-100 and Other Micro Buses, Howard W. Sams & Co., Inc.: Indianapolis, 1979, 144p.

Rationale for the Design of the Ada Programming Language, SIGPLAN Notices, 14:6, June 1979, part B.

Rattner, J., and Lattin, W., "Ada Determines Architecture of 32-bit Microprocessor", Electronics, February 24, 1981, pp.119-126.

Redell, D., et al, "Pilot: An Operating System for a Personal Computer", CACM 23:2, Feb. 1980, pp. 81-91.

Reference Manual for the Ada Programming Language, 1980.

Ritchie, D., "A Retrospective", Bell System Technical Journal 57:6, part 2, pp. 1947-1970.

Ritchie, D., and Thompson, K., "The UNIX Time-Sharing System", Bell System Techincal Journal 57:6, part 2, pp. 1905-1930.

Roberts, T., Evaluation of Computer Text Editors, Xerox Technical Report SSL-79-9, November 1979.

Robertson, C., et al, Experimental Evaluation of the ZOG Frame Editor, Carnegie Mellon University, CMU-CS-81-112, April 1981, 14p.

Roman, A., "Winchester Market Shifts to 5 1/4-in Drives", Mini-Micro Systems 14:2, February 1981, pp. 85-97.

Rosen, B., "High-Level Data Flow Analysis", CACM 20:10, Oct. 1977, pp. 712-724.

Saltzer, J., "Runoff", in Crisman, P., The Compatible Time-Sharing System, MIT Press: Cambridge, Mass., 1965.

Sandewall, E., "Programming in the Interactive Environment: The LISP experience" ACM Computing Surveys 10:1, March 1978, pp. 35-72.

Santoni, A., "Microcomputer Development Systems", EDN 25:16, September 5, 1980, pp. 141-150.

SDL/SSL, United Kingdom Ministry of Defense Ada Support System Study, Phase 2 & 3 Reports.

SDL/SSL, Ada Support System Study, Phase 4 Report, The Initial Host, June 180.

Schneiderman, B., Software Psychology, Winthrop, Cambridge, Mass., 1980, 320p.

Shimasaki, M., et al, "An Analysis of Pascal Programs in Compiler Writing", Software Practice and Experience 10:2, pp. 149-157.

Shugart Associates, SA400 Minifloppy Diskette Storage Drive OEM Manual, Revision 5, 1977.

Sites, R.L., "Programming Tools: Statement Counts and Procedure Timings", SIGPLAN Notices, 13:12, Dec. 1978, pp. 98-101.

Snodgrass, R., "A Sophisticated Microcomputer User Interface", in Proceedings of the Third Symposium on Small Systems, SIGSMALL, Sept. 1980, pp. 97-105.

Softech, Ada Compiler Validation Implementator's Guide.

Softech, Survey and Measurement of Properties of High-Level Lanaguage, AFWAL Interim Technical Report HLLM #4.14.1A, November 1980, 249p.

Sorcim, Pascal/M User's Reference Manual, Digital Marketing, Walnut Creek, Ca, 1979, 77p.

Stenning, V., "The Ada Environment: A Perspective", Computer 14:6, June 1981, pp.26-36.

Stonebraker, M., "Operating System Supports for Database Management", CACM 24:7, July 1981, pp. 412-418.

"Stoneman", Requirements for Ada Programming Support Environments.

Supersoft, Forth, Champaign, Il. 1980.

Suzuki, N., and Ishihata, K., "Implementations of an Array Bound Checker", POPL, 1977, pp. 132-143.

Szymanski, J., "Assembling Code for Machines with Span-Dependent Instructions", CACM 21:3, April 1978, pp. 300-308.

Tanenbaum, A., "Implications of Structured Programming for Machine Architecture, CACM 21:3, March 1978, pp. 237-246.

Teitelbaum, T., Reps, T. and Horwitz, S., "The Why and Wherefore of the Cornell Program Synthesizer, SIGPLAN 16:6, June 1981, pp. 8-16.

Teitelman, W. and Masinter, L., "The Interlisp Programming Environment", Computer 14:4, April 1981, pp. 25-34.

Texas Instruments, Ada Integrated Environment: Computer Program Development Specification, 1981.

Texas Instruments, Ada Integrated Environment: Interim Techincal Report, 1981.

Texas Instruments, Ada Integrated Environment: System Specification, 1981.

Texas Instruments, 990 Computer Family Systems Handbook, May 1976.

Thacker, C., et al, Alto: A Personal Computer, Xerox Technical Report CSL-79-11.

Thompson, K. "UNIX Implementation", Bell System Technical Journal, 57:6, part 2, July-August 1978, pp. 1931-1946.

Titus, C., et al, 16-Bit Microprocessors, Howard W. Sams & Co., Inc.: Indianapolis, 1981, 350p.

Tobias, J., "LSI/VLSI Building Blocks", Computer 14:8, August 1981, pp.83-101.

Toong, H. and Gupta, A., "An Architectural Comparison of Contemporary 16-Bit Microprocessors", IEEE Micro 1:2, May 1981.

TRW, User's Manual for Jovial Interactive Debugger, April 21 1981.

UCSD Pascal System Reference Manual, 1978.

Ullman, J., Principles of Database Systems, Computer Science Press, Potomac, Md., 1980, 379p.

Wadlow, T., "The Xerox Alto Computer", BYTE 6:9, September 1981, pp. 58-68.

Wallace, B., Microsoft Pascal Reference Manual Preliminary Version 1.8, Microsoft, Bellevue, Wa., 1979, 119p.

Wasserman, A. "Guest Editor's Introduction: Automated Deveopment Environments", Computer 14:4, April 1981, pp. 7-11.

Welsh, J., "Economic Range Checks in Pascal", Software-Practice & Experience 8:1, Jan. 1978, pp. 85-98.

Western Digital, WD/90 Pascal MICROENGINE Reference Manual.

Wirth, N., Algorithms + Data Structures = Programs, Prentice-Hall, 1976, 366p.

Wirth, N., "The Design of a PASCAL Compiler", Software-Practice & Experience 1:4, Oct.-Dec. 1971, pp. 309-334.

Wise, Chen and Yokely, Microcomputers: A Technology Forecast and Assessment to the Year 2000, John Wiley & Sons, New York, 1980, 251p.

Wolfe, M., et al, "The Ada Language System", Computer 14:6, June 1981, pp. 37-46.

Wulf, W., et al, The Design of an Optimizing Compiler, American Elsevier, New York, 1975, 165p.

Zaks, R., The CP/M Handbook with MP/M, Sybex: Berkeley, 1980. 321p.

Zaks, R., Military Microprocessor Systems, Sybex, 1976.

Zeigler, S., et al, "Ada for the Intel 432 Microcomputer", Computer 14:6, June 1981, pp. 47-56.

Zilog, Z8000 CPU Technical Manual, 1980.

Appendix A

Minimal J73/I Compiler Phase Sizes (in words) in memory

Host:	DEC-10		IBM		Univac 1108	
	Root size = 14,473		Root size = 17,352		Root size = 20,928	
Phase Name	Phase size	Total	Phase size	Total	Phase size	Total
J73	10,986	25,759	13,262	30,614	17,936	38,864
Pass 2	32,843	47,616	47,545	64,897	35,085	56,010
Pass 3	38,412	53,185	48,117	65,469	39,423	60,349
Pass 4	19,392	34,165	46,700	64,052	23,740	44,666
Pass 5	2,297	17,010	3,013	20,365	9,446	30,372

Minimal J73 Compiler Phase Sizes (in words) in memory

Host:	DEC-10		IBM		
	Root size = 17,418		Root size = 20,576		
Phase Name	Phase size	Total	Phase size	Total	
J73	13,834	31,252	17,698	38,274	
Pass 2A	43,622	61,040*	60,852	81,428	= 325,712 bytes
Pass 2B	42,395	59,813	59,237	79,813	
Pass 3	37,782	55,200	50,340	70,916	
Pass 4	19,722	37,140	50,026	70,602	
Pass 5	2,518	19,936	3,492	24,068	

Notes: * - this is 2,197,440 bits which is equivalent to 274,680 8-bit bytes.

Root size includes the EXEC and debugging routines which are permanently resident. Some of this space is used for the symbol table when the compiler is not being run in debugging mode.

Appendix B

Sizes of Executables on Disk for the SEA J73 Compiler on the DEC-10

Target	DEC-10	15A	HBC	MAGIC
Front End + compool output (Disk blocks)	1,340	1,340	1,340	1,340
Back End (Disk blocks)	736	740	704	800
Total Blocks	2,076	2,080	2,044	2,140
Size in Words (128 words/block)	265,728	266,240	261,362	273,920
Size in bits	9,566,208	9,584,640	9,418,752	9,061,120
Size in 8 bit byte equivalents	1,195,776	1,198,080	1,177,344	1,232,640

Notes: 1. The targets are:
HBC - AN/AYK-15
15A - MIL-STD-1750
MAGIC - DELCO M362F-2

2. The size listed under each target is for an individual target. Only one front end is required for all targets. Thus, for all four targets the total size (in blocks) is (1340 + 736 + 740 + 704 + 800) = 4320 rather than (4 * 1340 + 736 + 740 + 704 + 800).

3. Since the DEC-10 has 36 bit words and uses 7 bits/character, the sizes were computed to 8-bit byte equivalents because microcomputer storage is normally measured in 8-bit bytes.

Appendix C

J73/I Compilation Speeds

Compilation Speeds for J73/I compiler on DEC-10 (KL10) TENEX

Module	Lines	CPU Time (sec)	Lines/ CPU Minute	Elapsed Time (min)	Lines/ Minute	Notes
COGEN	3998	45.33	5289	13.5	296	No listings
REPL	3750	49.81	4517	15.6	240	
SDBUG	2993	26.74	6716	2.8	1069	
COMPUT	4087	47.33	5181	4.7	870	
COMPAR	3133	27.78	6767	3.9	803	
GPRCS	5749	80.58	4280	24.5	2350	
GPRCS	5749	80.08	4370	2.5	2300	Source
GPRCS	5749	106.32	3244	4.7	1223	Source + XREF

Compilation Speeds for J73/I Compilations on DEC-10 KI10

Module	Lines	CPU Time (sec)	Lines/CPU Minute	Elapsed Time (sec)	Lines/ Minute	Lines	CPU Time	Lines/CPU Minute
PEXPR	1393	23.611	3540	35	2388	1394	23.329	3585
KNTZTM						1826	29.908	3663
FPRT	1023	19.531	3143	44	1395	1019	17.904	3415
INIT	1541	36.210	2553	54	1712	1533	34.691	2651
INTCST	1715	29.574	3479	94	1095	1647	27.633	3576
INTRP	2364	51.954	2730	139	1020	2292	48.406	2841
ITMDSC	1765	30.700	3450	60	1765	1728	29.014	3573
ITMPRC						1793	29.439	3654
JCOM	873	17.152	3054	27	1940	869	16.422	3175
JOVDMD*	54	5.853	554	13	249	54	5.532	586
MYPRCS	2805	57.734	2915	124	1357	2804	55.977	3006
OTPK	1963	36.240	3250	86	1370	1917	33.512	3432
OVRPRC	2009	35.033	3441	120	1005	2008	33.793	3565
PRCPRI	1911	33.596	3413	80	1433	1960	33.671	3493
PRCPR2	1800	32.930	3280	60	1800	1799	30.600	3527
PRSETS						2052	36.048	3415
PRSET1	1849	30.920	3588	70	1585			
PRSET2	2486	45.456	3281	151	988			
SRCH	1063	19.877	3209	72	886	1057	17.344	3657
STATPR	1721	30.259	3413	90	1147	1722	28.982	3565

Module	Lines	CPU Time (sec)	Lines/CPU Minute	Elapsed Time (sec)	Lines/ Minute	Lines	CPU Time	Lines/CPU Minute
STRTN	1556	31.026	3009	48	1945	1550	29.855	3115
TBLPRC	2566	43.610	3530	63	2444	2017	32.776	3672
TOKN1	2543	50.280	3035	83	1838	2542	48.758	3128
TOKN2	1619	27.691	3514	43	2259	1618	26.9[illegible]	3607

Note: The compile speed for JOVDMD is much slower for two reasons. First, it is a much smaller module than the rest, so that phase load time is more significant. Second, the module contains many define expansions (more than one per line) so there is a significant amount of additional processing per source line.

JOVIAL/Ada Microprocessor Study Final Technical Report

Appendix D

Speeds for J73 Compilations of JCVS tests on DEC-10s

Name	Lines	KA10 time	KA10 speed	KI10 time	KI10 speed
ETEST1	2872	202	853	86	2004
ETEST2	4927	311	951	117	2527
ETEST3	2729	177	925	69	2373
ETEST4	3144	154	1225	74	2549
ETEST5	2788	140	1195	59	2835
ETEST6	2647	136	1168	59	2692

Notes: 1. Times are given in CPU seconds.
2. Speeds are given in lines per CPU second.

Appendix E

Compilation Times and Speeds for USCD Pascal Compilers

Name	Lines	Time	Lines/Minute	Notes
SRCH	162	27.8	349	No listings, No swapping
GSTMP	127	27.1	281	
FORM	381	1:00	381	
DASSY	957	3:26.8	278	
DASSY	984	3:33.4	277	
TRACP	202	1:09.2	175	No listings Compiled in swapping mode because there was not enough stack space to compile without swapping.
SRCH	164	1:27.6	112	Source listing to disk, no swapping.
GSTMP	127	1:10.3	108	
DASSY	957	13:34.8	70	
DASSY	985	13:46.0	72	
SRCH	164	1:42.1	96	Source listing to disk, swapping.
GSTMP	127	1:26.4	88	
TRACP	202	2:11.4	92	

Appendix F

Compilation, Assembly and Linking Times and Speeds for Pascal/Z

Name	Lines	CT	AT	C&AT	LT	TT	CS	C&AS	CLAS	Notes
SRCH	162	1:15.9	1:04.3	2:20.2	31.1	2:51.3	128	69	57	No listings
GSTMP	127	1:06.6	1:02.1	2:08.7	30.8	2:39.5	114	59	48	
TRACP	202	1:38.9	1:20.8	2:59.7	30.9	3:30.6	123	67	58	
SRCH	162	1:33.5	1:04.3	2:37.8	31.1	3:08.9	104	62	51	Source list only
GSTMP	127	1:22.1	1:02.1	2:24.2	30.8	2:50.0	93	53	45	
TRACP	202	1:57.7	1:20.8	3:18.5	30.9	3:49.4	103	61	53	
SRCH	162	1:33.5	3:28.7	5:02.2	31.6	5:33.8	104	32	29	All listings
GSTMP	127	1:22.1	2:05.9	3:28.9	36.3	4:05.2	93	36	31	
TRACP	202	1:57.7	3:26.0	5:23.7	35.5	5:59.2	103	37	34	

Notes: 1. These results were obtained through two sets of runs - one with listings and one without, rather than 3 independent sets of tests.
2. All speeds are in lines/minute.

Column Heading Key:
CT Compile Time
C&AT Compile & Assembly Time
LT Link Time
TT Total Time
CS Compile Speed
C&AS Compile & Assembly Speed
CLAS Compile, Link, Assembly Speed

Appendix G

Code Sizes for Compiler Benchmarks (in bits)

	Compiler			
	J73/I (DEC-10)	UCSD Pascal		Pascal Z
Test		Without jump table	With jump table	
SRCH	6,802 bits	2,536 bits	2,656 bits	6,336 bits
GSTMP	6,624 bits	3,096 bits	3,216 bits	7,432 bits
TRACP	11,340 bits	4,512 bits	4,640 bits	11,906 bits
Total	24,768	10,144	10,512	24,824
% of DEC-10 Size	100%	41%	42%	100%

Appendix H

Frequency of Occurrence of IL Tokens for Various Compiler Phases
(Statistics are for phases hosted on the IBM 370 unless otherwise stated)

J73/I Front End

IL	Occurrences	%
PRIM	38,912	41.39
STNO	10,551	11.22
ATTR	6,729	7.16
MUL	5,698	6.06
REPL	5,462	5.81
SUBS	5,049	5.37
LABL	4,157	4.42
PARM	4,128	4.39
CALL	2,197	2.34
ENDC	2,197	2.34
GOTO	1,897	2.02
NQ	1,136	1.21
AT	1,093	1.16
EQ	1,060	1.13
ADD	993	1.06
SUB	422	.45
BIT	295	.31
DIV	243	.26
PLST	213	.23
PROC	211	.22
EPRC	211	.22
GR	176	.19
LS	117	.12
LOC	116	.12
MNUS	98	.11
GQ	81	.09
LQ	79	.08
MOD	68	.07
BSIZ	61	.06
COMP	57	.06
UNKN	52	.06
AND	40	.04
OR	37	.04

J73 Front End
Host = DEC-10

IL	Occurrences	%
PRIM	51,511	40.08
STNO	15,322	11.92
ATTR	11,110	8.64
REPL	8,210	6.39
SUBS	7,463	5.81
PARM	7,116	5.54
LABL	6,900	5.37
CALL	3,547	2.76
ENDC	3,547	2.76
GOTO	2,644	2.06
AT	1,827	1.42
EQ	1,771	1.38
NQ	1,655	1.29
ADD	1,370	1.06
MUL	805	.63
SUB	599	.47
BIT	367	.29
PROC	334	.26
EPRC	334	.26
DIV	320	.25
PLST	213	.17
GR	197	.15
BSIZ	184	.14
MNUS	164	.13
LS	163	.13
LOC	131	.10
GQ	117	.09
LQ	97	.08
MOD	80	.06
UNKN	70	.05
AND	62	.05
COMP	43	.03
CIFR	40	.03

TERM	34	.04	OR	39	.03
SHIFT	33	.04	TERM	39	.03
FSIZ	32	.03	FSIZ	37	.03
CIFR	25	.03	SHIFT	36	.03
ABS	19	.02	ABS	28	.02
EXP	9	.01	EXP	16	.01
BYT	7	.01	CRIF	10	.01
CRIF	6	.01	BYT	8	.01
CSIZ	5	.01	XOR	6	.00
XOR	2	.00	CSIZ	2	.00
EQV	1	.00	EQV	2	.00
			FLAG	1	.00
	94,009			128,527	

Optimizer

IL	Occurrences	%
PRIM	10,973	41.41
STNO	2,959	11.17
ATTR	1,614	6.09
MUL	1,609	6.07
REPL	1,550	5.85
SUBS	1,404	5.30
LABL	1,326	5.00
PARM	1,183	4.46
CALL	601	2.27
ENDC	601	2.27
GOTO	582	2.20
AT	505	1.91
EQ	365	1.38
NQ	227	.86
ADD	157	.59
PROC	129	.49
EPRC	129	.49
SUB	111	.42
PLST	106	.40
BIT	43	.16
GQ	43	.16
LS	40	.15
MNUS	40	.15
COMP	28	.11
GR	26	.10
DIV	25	.09
AND	24	.09
LQ	21	.08
FSIZ	19	.07
MOD	12	.05
BSIZ	11	.04
OR	11	.04
LOC	11	.04
CIFR	8	.03
ABS	8	.03
SHIFT	6	.02
CRIF	3	.01
TERM	3	.01
XOR	2	.01
CSIZ	2	.01
BYT	1	.00
EQV	1	.00
	26,519	

Code Generator

IL	Occurrences	%
PRIM	25,861	40.58
ATTR	7,219	11.33
STNO	5,407	8.48
PARM	4,506	7.07
REPL	3,509	5.51
MUL	2,970	4.66
AT	2,794	4.38
SUBS	2,443	3.83
LABL	1,911	3.00
CALL	1,842	2.89
ENDC	1,842	2.89
GOTO	707	1.11
NQ	570	.89
EQ	557	.87
ADD	319	.50
PROC	179	.28
EPRC	179	.28
BIT	172	.26
PLST	138	.22
DSIZ	88	.14
AND	80	.13
SUB	63	.10
BSIZ	55	.09
LQ	54	.08
GR	53	.08
COMP	39	.06
LS	30	.05
MNUS	30	.05
OR	27	.04
DIV	21	.03
GQ	19	.03
MOD	14	.02
SHIFT	9	.01
TERM	8	.01
ABS	5	.01
LOC	3	.00
SPTR	3	.00
UNKN	1	.00
CSIZ	1	.00
	63,728	

Editor

IL	Occurences	%
PRIM	14,339	42.11
STNO	3,089	9.07
ATTR	2,298	6.75
MUL	2,030	5.96
PARM	2,001	5.88
REPL	1,838	5.40
SUBS	1,784	5.24
LABL	1,387	4.07
AT	944	2.77
CALL	815	2.39
ENDC	815	2.39
GOTO	626	1.84
ADD	474	1.39
NQ	281	.83
EQ	254	.75
BSIZ	140	.41
BIT	133	.39
SUB	126	.37
EPRC	91	.27
PROC	91	.27
LS	73	.21
PLST	58	.17
DIV	46	.14
GR	45	.13
GQ	37	.11
UNKN	37	.11
LOC	36	.11
COMP	31	.09
AND	22	.06
LQ	22	.06
CIFR	14	.04
BYT	14	.04
ABS	12	.04
OR	11	.03
CSIZ	10	.03
MNUS	10	.03
MOD	7	.02
TERM	5	.01
CRIF	2	.01
FSIZ	2	.01
SHIFT	1	.00
	34,051	

TOTALS
J73/I

IL	Occurences	%
PRIM	90,085	41.27
STNO	22,006	10.08
ATTR	17,860	8.18
REPL	12,359	5.66
MUL	12,307	5.64
PARM	11,818	5.41
SUBS	10,680	4.89
LABL	8,781	4.02
CALL	5,455	2.50
ENDC	5,455	2.50
AT	6,070	2.51
GOTO	3,812	1.75
EQ	2,236	1.02
NQ	2,214	1.01
ADD	1,943	.89
SUB	722	.33
BIT	715	.29
PROC	610	.28
EPRC	610	.28
PLST	515	.24
DIV	335	.15
GR	300	.14
BSIZ	267	.12
LS	260	.12
GQ	180	.08
MNUS	178	.08
LQ	176	.08
LOC	166	.08
AND	166	.08
COMP	155	.07
MOD	101	.05
UNKN	90	.04
DSIZ	88	.04
OR	86	.04
FSIZ	53	.02
TERM	50	.02
SHIFT	49	.02
CIFR	47	.02
ABS	44	.02
BYT	22	.01

TOTALS
J73
Host = DEC-10

IL	Occurences	%
PRIM	97,053	40.18
STNO	26,787	11.08
ATTR	22,241	9.21
REPL	15,107	6.25
PARM	14,806	6.13
SUBS	13,094	5.42
LABL	11,524	4.77
CALL	6,805	2.82
ENDC	6,805	2.82
AT	5,336	2.44
GOTO	4,559	1.89
EQ	2,947	1.22
NQ	2,733	1.13
ADD	2,310	.96
MUL	1,783	.74
SUB	899	.37
PROC	733	.30
EPRC	733	.30
BIT	715	.30
PLST	515	.21
DIV	412	.17
BSIZ	390	.16
GR	321	.13
LS	306	.13
MNUS	244	.10
GQ	216	.09
LQ	194	.08
AND	188	.08
LOC	181	.07
COMP	141	.06
MOD	113	.05
UNKN	108	.04
DSIZ	88	.04
OR	88	.04
CIFR	62	.03
FSIZ	58	.02
TERM	55	.02
ABS	53	.02
SHIFT	52	.02
BYT	23	.01

CSIZ	18	.01	EXP	16	.01
CRIF	11	.01	CSIZ	15	.01
EXP	9	.00	CRIF	15	.01
XOR	4	.00	XOR	8	.00
SPTR	3	.00	EQV	3	.00
EQV	2	.00	SPTR	3	.00
			FLAG	1	.00
	218,307			241,563	

Appendix I

Frequency of Occurrence for Statements in Compiler Phases

J73/I Front End

Statement	Occurrences	%
Assignment	4,079	44.92
If	2,196	24.19
Proc call	1,410	15.53
Goto	1,004	11.06
Return	160	1.76
For	152	1.67
Switch	31	.34
Mult. Assign.	30	.33
While	18	.20

J73 Front End

Statement	Occurrences	%
Assignment	5,706	45.93
If	2,793	22.48
Proc call	2,219	17.86
Goto	1,078	8.68
Return	251	2.02
For	205	1.65
While	70	.56
Switch	55	.44
Mult. Assign.	47	.38

Optimizer

Statement	Occurences	%
Assignment	896	40.88
If	478	21.81
Proc call	394	17.97
Goto	291	13.28
For	73	3.33
Return	38	1.73
Switch	17	.78
While	3	.14
Mult. Assign.	2	.11

Code Generator

Statement	Occurences	%
Proc call	1584	39.48
Assignment	1299	32.38
If	826	20.59
Goto	141	3.51
Return	72	1.79
For	56	1.40
Switch	27	.67
While	5	.12
Mult. Assign.	2	.05

Editor

Statement	Occurences	%
Assignment	1,061	40.59
Proc call	764	29.23
If	517	19.78
Goto	122	4.67
For	85	3.20
Return	36	1.38
Switch	24	.92
While	4	.15
Mult. Assign.	1	.04

J73/I TOTALS

Statement	Occurences	%
Assignment	7,335	40.98
Proc call	4,152	23.20
If	4,017	22.44
Goto	1,558	8.70
For	366	2.04
Return	306	1.71
Switch	99	.55
Mult. Assign.	35	.20
While	30	.17
	17,898	

J73 TOTALS

Statement	Occurences	%
Assignment	8,962	42.19
Proc call	4,961	23.35
If	4,614	21.72
Goto	1,632	7.68
For	419	1.97
Return	397	1.87
Switch	123	.58
While	82	.39
Mult. Assign.	52	.24
	21,242	

Note: 1. The percentages shown under the heading "J73 TOTALS" were used as a basis for the mix figures (Chapter 19, Appendices L and M).

Appendix J

Execution benckmark results
(Times in Seconds)

Test	I	C	T	P	A	KA	KI	KL
Ackerman	207.0	75.1			18.7	27.	11.8	3.2
*	199.3	62.7				24.7	10.6	3.0
Sieve	18.4	11.3			4.4	1.4	.47	.16
*	15.9	2.7				.97	.35	.10
Prime	256.3	746.8						

Quicksort
(recursive)
With bounds
checking

Elements	I	C	T	P	A	KA	KI	KL
50	.4							
100	.9	.4						
500	6.5	3.1						
1000	14.2	6.6			3.4	1.3	.55	.15
1000+						1.1	.39	.11
5000	85.8	40.6		17.2	21.7	9.4	3.0	.90
5000+						6.3	2.3	.79

No bounds
checking

Elements	I	C	T	P	A	KA	KI	KL
50	.3		.4					
100	.6	.3	.9					
500	4.8	1.0	6.5					
1000	10.9	2.3	14.2			1.2	.40	.13
1000+						.8	.28	.07
5000	63.6	14.4	83.5	3.2		6.7	2.3	.69
5000+						4.5	1.7	.46

Quicksort
(Iterative)

Elements	I	C
50	.4	
100	.9	.4
500	6.8	3.3
1000	14.6	7.3
5000	87.6	43.2

Bubblesort
With bounds
checking

Elements	I	C
50	1.9	.8
100	7.6	3.6
500	186.3	90.3
1000		361.2

Bubblesort
No bounds
checking

Elements	I	C	T
50	1.3	.3	1.8
100	5.8	1.1	7.4
500	145.8	30.8	180.6
1000		122.9	

Key:

I = Interpretive - UCSD Pascal on Northstar Horizon (4 Mhz Z80)

C = Compiled - Pascal/Z on Horizon

T = Threaded - SL5 on Horizon

P = Pascal - Telesoft Pascal on 8Mhz 68000 with Q-bus

A = Ada - Telesoft Ada on 68000

KA = DEC-10 KA10

KI = DEC-10 KI10

KL = DEC-10 KL10

Notes: 1. Rows marked with an asterisk represent speeds with range checking turned off, except for the sort tests, where checking is indicated in the heading.

2. For the sort tests rows marked with a plus sign represent times with the main array allocated statically (i.e., not on the runtime stack). Other figures are for the array on the stack.

Execution benchmark results
(Normalized speeds)

Test	I	C	T	P	A	KA	KI	KL
Ackerman	.4	1.0			4.0	2.8	6.4	23
*	.3	1.0				2.5	5.9	21
Sieve	.6	1.0			2.6	8.1	24	71
*	.2	1.0				1.9	5.7	27
Prime	3.0	1.0						

Quicksort
(recursive)
With bounds
checking

Elements	I	C	T	P	A	KA	KI	KL
50								
100	.4	1.0						
500	.5	1.0						
1000	.5	1.0			1.9	5.1	12	44
+						6	17	60
5000	.5	1.0		2.4	1.9	4.3	14	45
+						6.4	18	51

No bounds
checking

Elements	I	C	T	P	A	KA	KI	KL
50								
100	.5	1.0	.3					
500	.2	1.0	.2					
1000	.2	1.0	.2			1.9	5.8	18
+						2.9	8.2	33
5000	.2	1.0	.2	4.5		2.1	6.3	21
+						3.2	8.5	31

Quicksort
(Iterative)

Elements	I	C
50	.4	1.0
100	.4	1.0
500	.5	1.0
1000	.5	1.0
5000	.5	1.0

Bubblesort
With bounds
checking

Elements	I	C
50	.4	1.0
100	.5	1.0
500	.5	1.0
1000		1.0

Bubblesort
No bounds
checking

Elements	I	C	T
50	.3	1.0	.2
100	.2	1.0	.1
500	.2	1.0	.2
1000		1.0	

Key:

I = Interpretive - USCD Pascal on Northstar Horizon (4Mhz Z80)

C = Compiled - Pascal/Z on Horizon

T = Threaded - SL5 on Horizon

P = Pascal - Telesoft Pascal on 8Mhz 68000 with Q-bus

A = Ada - Telesoft Ada on 68000

KA = DEC-10 KA10

KI = DEC-10 KI10

KL = DEC-10 KL10

Appendix K

Compilation Statistics

	Lines	State-ments	Symbol Table Size (1)	Symbol Table Size (2)	Number of Entries	Bytes/ entry
GPRCS	5797	4824	237,868	90,256	5311	17
REPL	3819	2704	180,732	57,776	3529	16
COMPUT	4095	2784	183,084	59,876	3645	16
EDIT	3555	1931	151,108	49,072	3020	16

Notes:
1. This size is in bytes and includes debugging routines as well as name and define strings.
2. This size is the number of bytes for symbol table entries other than names and defines. (The debugging routines included in Size (1) are excluded here.)

	Names	Defines	IL size (bytes)	IL bytes/ statement
GPRCS	2215	294	65,972	14
REPL	2109	295	35,380	13
COMPUT	2161	298	35,736	13
EDIT	1929	172	29,164	15

Appendix L

Compiler Gibson Mix - Space

Machine	Mix	z	a	c	p	f	g	i
P	1	4.4	3.5	3.8	2.0	1.7	3.0	6.7
	2	4.4	3.5	3.8	2.0	1.7	3.0	6.7
	3	5.7	5.1	5.3	2.0	1.8	3.0	8.1
	4	5.7	5.1	5.3	2.0	1.8	3.0	8.1
P*	1	5.5	4.3	5.5	2.0	1.9	3.0	8.2
	2	5.5	4.3	5.5	2.0	1.9	3.0	8.2
	3	6.9	6.0	7.2	2.0	2.0	3.0	9.7
	4	6.9	6.0	7.2	2.0	2.0	3.0	9.7
Z80	1	13.	12.	15.	25.	10.	3.0	14.
	2	12.	12.	15.	25.	10.	3.0	14.
	3	15.	14.	17.	25.	11.	3.0	16.
	4	14.	14.	17.	25.	11.	3.0	16.
8086	1	7.4	6.0	9.3	11.	12.	3.0	8.3
	2	7.1	6.0	9.3	11.	12.	3.0	8.3
	3	8.7	7.5	11.	11.	13.	3.0	9.7
	4	8.4	7.5	11.	11.	13.	3.0	9.7
Z8000	1	10.	7.7	13.	25.	12.	4.0	9.5
	2	9.4	7.7	13.	25.	12.	4.0	9.5
	3	13.	11.	16.	25.	14.	4.0	12.
	4	12.	11.	16.	25.	14.	4.0	12.
MC68000	1	8.2	6.6	11.	11.	12.	2.0	9.4
	2	7.9	6.6	11.	11.	12.	2.0	9.4
	3	9.7	8.5	13.	11.	13.	2.0	10.
	4	9.5	8.5	13.	11.	13.	2.0	10.

Machine	Mix	z	a	c	p	f	g	i
DEC10	1	2.6	2.0	2.8	6.0	4.0	1.0	3.1
(words)	2	2.5	2.0	2.8	6.0	4.0	1.0	3.1
	3	3.1	2.6	3.4	6.0	4.4	1.0	3.7
	4	3.0	2.6	3.4	6.0	4.4	1.0	3.7
DEC10	1	12.	9.0	13.	27.	18.	4.5	14.
(8-bit	2	11.	9.0	13.	27.	18.	4.5	14.
bytes)	3	14.	12.	15.	27.	20.	4.5	17.
	4	14.	12.	15.	27.	20.	4.5	17.
IBM370	1	11.	7.9	13.	35.	20.	4.0	12.
	2	10.	7.9	13.	35.	20.	4.0	12.
	3	14.	11.	16.	35.	22.	4.0	14.
	4	13.	11.	16.	35.	22.	4.0	14.
VAX	1	6.3	4.9	9.3	6.0	8.0	2.0	7.2
	2	6.1	4.9	9.3	6.0	8.0	2.0	7.2
	3	7.6	6.4	11.	6.0	8.9	2.0	8.6
	4	7.4	6.4	11.	6.0	8.9	2.0	8.6
CAPS10	1	3.9	2.5	4.8	5.0	10.	2.0	5.0
	2	3.7	2.5	4.8	5.0	10.	2.0	5.0
	3	4.6	3.4	5.7	5.0	11.	2.0	5.9
	4	4.5	3.4	5.7	5.0	11.	2.0	5.9
CAPS10*	1	5.2	4.1	6.5	5.0	10.	2.0	6.6
	2	5.1	4.1	6.5	5.0	10.	2.0	6.6
	3	6.0	5.1	7.4	5.0	11.	2.0	7.4
	4	5.9	5.1	7.4	5.0	11.	2.0	7.4
1750A	1	6.7	4.4	9.3	19.	10.	2.0	6.2
	2	6.2	4.4	9.3	19.	10.	2.0	6.2
	3	7.7	5.7	11.	19.	11.	2.0	7.3
	4	7.2	5.7	11.	19.	11.	2.0	7.3

Machine	Mix	z	a	c	p	f	g	i
Nebula	1	6.4	4.9	8.5	10.	8.0	2.0	7.2
	2	6.1	4.9	8.5	10.	8.0	2.0	7.2
	3	6.9	5.5	9.1	10.	8.4	2.0	7.8
	4	6.6	5.5	9.1	10.	8.4	2.0	7.8
Data	1	1.4	1.8	1.8	0.0	1.0	0.0	1.5
	2	1.4	1.8	1.8	0.0	1.0	0.0	1.5
	3	1.7	2.1	2.1	0.0	1.2	0.0	1.8
	4	1.7	2.1	2.1	0.0	1.2	0.0	1.8

Statement Type Codes (from [Bloom74])

z	All statements
a	Assignment statements
c	Call statements
p	Procedure prologues, epilogues
f	For statements
g	Goto statements
i	If statements

Machine Codes

P	UCSD Pascal Pseudo - machine
Z80	Zilog Z80
8086	Intel 8086, iAPX86
Z8000	Zilog Z8001 (similar to DIS)
MC68000	Motorola MC68000
DEC10	Digital Equipment Corporation PDP-10
IBM370	IBM360, 370, 30XX and lookalikes
VAX	Digital Equipment Corporation VAX-11/780
CAPS7	Collins CAPS-7
CAPS10	Collins CAPS-10
1750A	MIL-STD-1750A
Nebula	MIL-STD-1862A
Data	This is not a machine, but represents the weighted average of the number of references to local data for the statements in the mix.

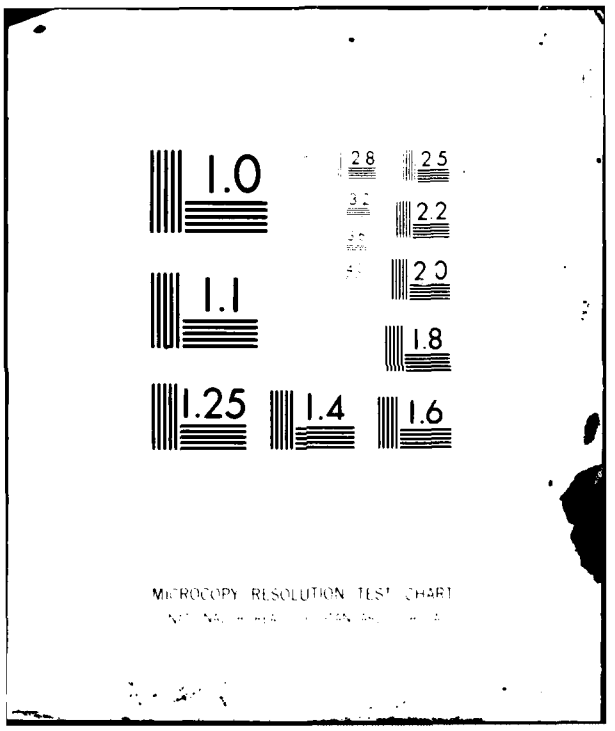
Mix Codes

1. Using statement mix from [Bloom74].
2. Using statement mix from Appendix I.
3. Using statement mix from [Bloom74] with subscripting correction based on statistics from Appendix F.
4. Using statement mix from Appendix I with subscripting correction based on statistics from Appendix H.

Notes: 1. All figures represent bytes, except as noted.

2. Two sets of figures are given for the P machine and CAPS. The first set represent the best-case assumption (i.e., all data may be referenced with the shortest possible instruction). For these machines this assumption only holds if there are fewer than 16 local data items. The figures marked with the asterisks represent a more reasonable assumption, up to 256 data items.

AD-A116 352 SOFTWARE ENGINEERING ASSOCIATES INC TORRANCE CA F/G 9/2
JOVIAL/ADA MICROPROCESSOR STUDY.(U)
APR 82 T E DEVINE, T L DUNBAR, M B LITTLEJOHN F30602-80-C-0153
UNCLASSIFIED RADC-TR-82-61 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
MICROCOPY RESOLUTION TEST CHART

Appendix M

Compiler Gibson Mix - Time

Machine	Mix	z	a	c	p	f	g	i
P	1	770	390	1800	0	1200	160	570
	2	760	390	1800	0	1200	160	570
	3	980	640	200	0	1300	160	780
	4	780	390	1770	0	1300	160	570
	5	760	390	1770	0	1300	160	570
	6	990	640	2010	0	1450	160	780
Z80	1	110	70	100	170	30	12	70
	2	110	70	100	170	30	12	70
	3	130	85	120	170	40	12	85
	4	110	70	100	170	34	12	70
	5	110	70	100	170	34	12	85
	6	130	85	120	170	44	12	85
8086	1	55	30	75	50	15	15	40
	2	50	30	75	50	15	15	40
	3	60	38	80	50	20	15	45
	4	55	30	75	50	25	15	40
	5	50	30	75	50	25	15	40
	6	60	38	80	50	30	15	45
Z8000	1	45	21	45	80	18	6	25
	2	44	21	45	80	18	6	25
	3	55	30	55	80	23	6	30
	4	45	21	45	80	23	6	25
	5	44	21	45	80	23	6	25
	6	55	30	55	80	28	6	30

Machine	Mix	z	a	c	p	f	g	i
MC68000	1	43	22	50	55	22	10	30
	2	42	22	50	55	22	10	30
	3	49	29	55	55	26	10	35
	4	43	22	50	55	30	10	30
	5	42	22	50	55	30	10	30
	6	49	29	55	55	34	10	35
KA10	1	9.9	5.3	8.7	16	7.2	1.8	6.0
	2	9.9	5.3	8.7	16	7.2	1.8	6.0
	3	11	7.2	11	16	8.2	1.8	7.6
	4	9.9	5.3	8.7	16	5.7	1.8	6.0
	5	9.6	5.3	8.7	16	5.7	1.8	6.0
	6	11	7.2	11	16	8.2	1.8	7.6
KI10	1	6.1	2.6	5.1	11	5.3	1.4	3.9
	2	5.9	2.6	5.1	11	5.3	1.4	3.9
	3	6.8	3.4	5.9	11	5.8	1.4	4.6
	4	6.1	2.6	5.1	11	5.3	1.4	3.9
	5	5.9	2.6	5.1	11	5.3	1.4	3.9
	6	6.8	3.4	5.9	11	5.8	1.4	4.6
360/75	1	4.4	2.8	3.0	5.9	2.8	1.0	3.9
	2	4.4	2.8	3.0	5.9	2.8	1.0	3.9
	3	4.3	2.7	2.9	5.9	2.8	1.0	3.8
	4	4.4	2.8	3.0	5.9	1.3	1.0	3.9
	5	4.4	2.8	3.0	5.9	1.3	1.0	3.9
	6	4.3	2.7	2.9	5.9	1.2	1.0	3.8

Statement Type Codes (from [Bloom74])

Z	All statements
a	Assignment statements
c	Call statements
p	Procedure prologues, epilogues
f	For statements
g	Goto statements
i	If statements

Mix Codes

1. Using statement mix from [Bloom74], loops executed 0 times.
2. Using statement mix from Appendix I, loops executed 0 times.
3. Using statement mix from Appendix I, loops executed 0 times, subscripts.
4. Using statement mix from [Bloom74], loops executed 100 times.
5. Using statement mix from Appendix I, loops executed 100 times.
6. Using statement mix from Appendix I, loops executed 100 times, subscripts.

Additional Machine Codes (others are described in Appendix L)

KA10	DEC-10 KA10 Processor
KI10	DEC-10 KI10 Processor
360/75	IBM 360/75

Notes: 1. Times are given in clock cycles, except for the KA10, KI10 and 360/75, whose times are given in microseconds.
2. Times are from manufacturers' literature, except for the P machine.
3. P machine values were obtained by timing selected statements, solving linear equations to get times for individual operators and dividing out time per machine cycle, to convert to number of (Z80) machine cycles. No figure is given for procedure prologues and epilogues for the P machine. Times for these are included under c (calls).

Appendix N

List of Companies

The following is a list of addresses of companies which are mentioned in this report. It is not, by any means, a complete list of companies in the microcomputer or semiconductor industries.

Action Computer Enterprise, Inc.
55 West Del Mar Blvd.
Pasadena, CA 91105
(213) 793-2440

Advanced Micro Devices
901 Thompson Place
Sunnyvale, CA 94086
(408) 732-2400

Apollo Computer Inc.
5 Executive Park Drive
N. Billerica, MA 01862
(617) 667-8800

Apple Computer Inc.
10260 Bandley Dr.
Cupertino, CA 95014
(800) 538-9696
(800) 662-9238 (in CA)

Atari
1340 Bordeaux Ave.
Sunnyvale, CA 94086

Commodore Business Machines, Inc.
300 Valley Forge Square
681 Moore Road
King of Prussia, PA 19406

Computer Automation, Inc.
18651 Von Karman
Irvine, CA 92713
(714) 833-8830

Computex
5710 Drexel Ave.
Chicago, IL 60637
(312) 684-3183

Computhink
3260 Alpine Road
Menlo Park, CA 94025
(415) 854-2577

Digiac Corporation
175 Engineers Road
Smithtown, NY 11787
(516) 273-8600

Digital Electronic Systems, Inc.
Box 5252
Torrance, CA 90510
(213) 539-6239

Digital Research
P.O. Box 579
Pacific Grove, CA 93950
(408) 649-3896

Fairchild Semiconductor
464 Ellis Street
Mountain View, CA 94040
(415) 962-3278

Heath/Zenith
Heath Co.
Benton Harbor, MI 49022

Hewlett-Packard
Data Systems Division
11000 Wolfe Road
Cupertino, CA 95014

Intel Corp.
3065 Bowers Ave.
Santa Clara, CA 95051
(408) 987-8080

Ithaca Intersystems Inc.
1650 Hanshaw Road
P.O. Box 91
Ithaca, NY 14850
(607) 257-0190

MOS Technology, Inc.
950 Rittenhouse Road
Norristown, PA 19401

Motorola Semiconductor Products Inc.
3501 Ed Bluestein Blvd.
Austin, Texas 78721

National Semiconductor Corp.
2900 Semiconductor Drive
Santa Clara, CA 95051
(408) 737-5000

North Star Computers, Inc.
2547 Ninth Street
Berkeley, CA 94710
(415) 549-0858

Ohio Scientific
1333 South Chillicothe Road
Aurora, OH 44202
(216) 831-5600

OSM Computer Corporation
2364 Walsh Avenue
Santa Clara, CA 95051
(408) 496-6910

Radio Shack
1300 One Tandy Center
Fort Worth, TX 76102

RR Software
P.O. Box 1512
Madison, WI 53701
(608) 244-6436

TeleSoft
10639 Roselle Street
San Diego, CA 92121
(714) 457-2700

TeleVideo Systems Inc.
1170 Morse Ave.
Sunnyvale, CA 94086
(408) 745-7760

Texas Instruments Inc.
Semiconductor Group
P.O. Box 1443
Houston, Texas 77001

Vector Graphic Inc.
31364 Via Colinas
Westlake Village, CA 91362
(800) 423-5857
(800) 382-3367 (in CA)

Western Digital Corp.
3128 Redhill Ave.
Box 2180
Newport Beach, CA 92663
(714) 557-3550

Zilog, Inc.
10460 Bubb Road
Cupertino, CA 95014
(408) 446-4666

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


FILMED
8-8